Bachelorarbeit

Dynamic Analysis of Concurrent Go Programs

Erik Daniel Kassubek

Gutachter: Prof. Dr. Thiemann

Betreuer: Prof. Dr. Thiemann

Prof. Dr. Sulzmann

Albert-Ludwigs-Universität Freiburg

Technische Fakultät

Institut für Informatik

14. Februar 2023

Bearbeitungszeit

14. 11. 2022 – 14. 02. 2023

Gutachter

Prof. Dr. Thiemann

Betreuer

Prof. Dr. Thiemann

Prof. Dr. Sulzmann

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg i.Br., 23.02.2023

Ort, Datum	Unterschrift

Zusammenfassung

Go ist eine Programmiersprache, welche einen besonderen Fokus auf die Erzeugung von effizienten nebenläufigen Programmen legt. Dazu stellt Go Routinen als leichtgewichtige Threads sowie Mutexe und Channels zur Synchronisation von und Kommunikation zwischen solchen Routinen bereit. Solche Konstrukte können leicht zu Concurrency-Bugs führen, welche auf Grund ihre starken Abhängigkeit von der genauen Ausführung des Programms nur schwer erkennbar sind.

Diese Arbeit entwickelt und implementiert einen dynamischen Detektor zur automatischen Erkennung von blockenden oder terminierenden Concurrency-Bugs. Der Detektor verwendet dabei die Detektion von Bugs durch Mutexe mit Hilfe von Lock-Bäumen, sowie die Betrachtung von Bugs auf Channels mit Hilfe von Vector-Clocks. Außerdem ist der Detektor in der Lage mehrere, durch Select-Statement verzweigte Ausführungspfade zu betrachten.

Inhaltsverzeichnis

1. Einführung

Go ist eine seit 2007 von Google entwickelte Programmiersprache, welche sich in den letzten Jahren zu einer der beliebtesten Programmiersprachen entwickelt hat [1]. Ein Fokus der Sprache liegt dabei auf der Erstellung sicherer und effizienter nebenläufiger Programme. Go stellt verschiedene Funktionen für die Entwicklung solcher Programme zur Verfügung. Die wichtigsten dieser Funktionen sind dabei Go-Routinen, Mutexe und Channels.
Go-Routinen erlauben das nebenläufige Ausführen mehrerer Codeteile. Mutexe erlauben die Synchronisation von Routinen, sowie die Lösung des Problems des kritischen Ausschlusses. Channel können ebenfalls für die Synchronisation verwendet werden, es ist aber auch möglich mit ihnen Daten zwischen Routinen zu versenden. Programmierer werden dabei angehalten, Channel anstelle von globalen Variablen zu verwenden, da bei diesen die Wahrscheinlichkeit für das Auftreten von Concurrency-Bugs geringer ist [2].
Blockierende Concurrency-Bugs, welche durch die Nebenläufigkeit von Programmen sowie ihrer Synchronisationsmechanismen entstehen, sind in der Praxis aber dennoch sehr weit verbreitet [3].

Diese Arbeit beschäftigt sich mit der Entwicklung und Implementierung eines dynamischen Detektors “GoChan” für die automatische Erkennung von blockenden und terminierenden Bugs, welche durch die Verwendung von Mutexen und Channels in nebenläufigen Go-Programmen entstehen. Der Detektor vereinigt und erweitert dabei verschiedene Methoden aus iGoodLock [4] und UNDEAD [5] sowie Gopherlyzer-GoScout [6] und GFuzz [7] zur Erkennung und Analyse von durch Mutexe und Channels erzeugten Bugs.

GoChan besteht dabei aus zwei Programmen, einem Instrumenter und einem Analyzer. Für beide Programme existiert in

`https://github.com/ErikKassubek/GoChan`

eine Implementierung.
Mit dem Instrumenter wird der zu analysierende Programmcode so verändert, dass er im Anschluss analysiert werden kann. Für die Analyze wird das zu analysierende Programm mehrfach durchlaufen. Bei jedem Durchlauf zeichnet der Analyzer relevante Situationen auf und führt anschließend, basieren auf dem so erstellten Trace eine Analyse aus, um Probleme zu erkennen. Die Analyse wird mehrfach wiederholt, um verschiedene Ausführungspfade zu analysiere, welche durch die Verwendung verschiedener Cases in Select-Operationen erzeugt werden. Um den Trace eines Programmdurchlaufs aufzeichnen zu können und bei Select-Operationen verschiedene Ausführungspfade zu erzwingen, muss der zu analysierende Programmcode verändert werden. Dies kann durch den Instrumenter automatisiert erfolgen.

Die Arbeit ist folgendermaßen aufgebaut. Zuerst werden in Abschnitt 2 die Funktionsweise von Nebenläufigkeit in Go, sowie die zu detektierenden Bug-Situationen betrachtet. Anschließend wird in Abschnitt 3 die theoretische Funktionsweise des Detektors dargelegt. Im Abschnitt 4 und 5 werden die Funktionsweise des Instrumenters sowie die Implementierung des Detektors beschrieben. In Abschnitt 6 wird der Detektor auf konstruierte und tatsächliche Programme angewandt, um zu überprüfen, wie gut er in der Lage ist, Probleme richtig einzuschätzen. Für die tatsächlichen Programme werden dabei Programme aus GoBench [8] verwendet, welche eine Sammlung von Programmen mit Concurrency-Bugs aus mehreren großen open-source Programmen besitzt. Zuletzt werden die Erkenntnisse in Abschnitt 7 zusammengefasst.

2. Theorie

Im Folgenden sollen der theoretische Hintergrund bezüglich Nebenläufigkeit in Go betrachtet werden. Außerdem werden die durch den Detektor betrachteten Probleme aufgezeigt.

2.1. Routinen

Bei Go-Routinen handelt es sich um leichtgewichtige Threads, welche nebenläufig mit anderen Routinen auf dem selben Adressraum laufen [9]. Anders als Threads in vielen anderen Programmiersprachen werden sie nicht durch den OS-Kernel, sondern durch die Go-Runtime selber gesteuert. Diese besitzt einen eigenen Scheduler, welcher eine Technik namens m:n-Scheduling verwendet, um durch Multiplexing m Go-Routinen auf n OS-Threads auszuführen. Diese Herangehensweise hat den Vorteil, dass dadurch eine bessere Ausführungsgeschwindigkeit erreicht und ein geringerer Speicher benötigt wird. Dadurch ist es möglich, dass tausende oder sogar hunderttausende Routinen gleichzeitig in einem Programm laufen. Sie erleichtert außerdem die Implementierung von Synchronisations- und Kommunikationsmechanismen.

2.2. Mutexe

Mutexe (auch Locks genannt) gehören zu den am weitesten verbreitenten Mechanismen zur Synchronisierung nebenläufiger Programme [5]. Sie bilden eine Lösung für das Problem des

kritischen Abschnitts. In solch einem kritischen Abschnitt darf sich immer nur maximal eine Routine gleichzeitig aufhalten. Solche Abschnitte werden mit einem Mutex umschlossen. Sie werden unter anderem verwendet um zu verhindern, dass mehrere Routinen gleichzeitig auf die selbe globale Datenstruktur zugreifen. Mutexe besitzen zwei Zustände, geöffnet und geschlossen, mit denen ihre Verhalten gesteuert wird. Sie besitzen dabei die folgenden Operationen:

- Lock: Die Lock-Operation wird vor dem Eintritt in einen kritischen Bereich aufgerufen. Sie versucht den Mutex zu schließen. Ist es momentan geöffnet, wird der Mutex geschlossen und der kritische Bereich ausgeführt. Ist der Mutex bereits geschlossen, dann muss die Routine, in welcher die Lock-Operation ausgeführt werden soll so lange warten, bis der Mutex wieder geöffnet wird und geschlossen werden kann.

- TryLock: Eine TryLock Operation versucht, wie die Lock-Operation, einen Mutex zu schließen. Anders als bei Lock wird die Routine allerdings nicht blockiert, wenn eine Schließung nicht direkt möglich ist. Es wird lediglich zurückgegeben, ob sie erfolgreich war oder nicht. Ein Programmierer kann in diesem Fall selber entscheiden, wie das Programm weiter ablaufen soll, ob also z.B. der kritische Bereich übersprungen werden soll.

- Unlock: Diese Operation öffnet einen geschlossenen Mutex. Der Versuch ein geöffnetes Mutex zu öffnen führt zu einem Laufzeitfehler. Theoretisch verhalten sich Mutexe in Go wie binäre Semaphore, d.h. es ist möglich, dass eine Routine ein geschlossenes Mutex frei gibt, obwohl das Mutex von einer anderen Routine geschlossen wurde. Da dies in der Praxis sehr leicht zu einem unvorhersehbaren Verhalten führen kann, ist es fast immer üblich, ein Mutex immer nur von derjenigen Routine freizugeben, von der es beansprucht wurde. Im weiteren wird daher angenommen, dass ein Mutex immer von der Routine geöffnet wird, von welchem es geschlossen wurde.

Mutexe können mit (Try)RLock Operationen zu RW-Mutex erweitert werden. Dabei kann der selbe Mutex von mehreren (Try)RLock-Operationen geschlossen werden, ohne dass

es zwischenzeitlich geöffnet werden muss. Es ist allerdings nicht möglich, dass ein Mutex gleichzeitig über eine RLock- und eine Lock-Operation geschlossen wird. RW-Mutexe können z.B. verwendet werden, wenn mehrere Routinen gleichzeitig lesend auf eine Datenstruktur zugreifen dürfen, allerdings nur, wenn gerade keine Routine schreibend auf die selbe Datenstruktur zugreift.

2.3. Channel

Channel [9] ermöglichen es Routinen untereinander zu kommunizieren. Ein Channel `c` kann Daten `d` senden (`c <- d`) und empfangen (`<- c`). Das genaue Verhalten der Channel hängt dabei davon ab, ob es sich um einen gebufferten oder ungebufferten Channel handelt.
Bei einem ungebufferten Channel müssen Send- und Receive-Operation gleichzeitig ablaufen. Möchte eine Routine `R` Daten auf einem Channel `c` senden, ist aber keine Routine bereit die Daten von `c` zu empfangen, dann muss `R` so lange warten, bis eine andere Routine ein Receive-Statement auf dem Channel `c` ausführt. Das selbe gilt auch, wenn eine Routine an ein Receive-Statement eines Channels kommt auf welchem momentan nicht gesendet wird.
Bei einem gebufferten Channel der Größe n können bis zu n Nachrichten zwischengespeichert werden. Dies bedeutet, dass Send und Receive nicht mehr gleichzeitig ablaufen müssen. Eine Routine muss hierbei nur dann vor einer Operation warten, wenn eine Send-Operation auf einem vollen Channel, oder eine Receive-Operation auf einem leeren Channel ausgeführt wird. In allen anderen Fällen kann die Operation die Nachricht in den Buffer schreiben, bzw. eine Nachricht aus dem Buffer lesen.
Basierend auf dem Go-Memory-Model gilt, dass zum einen das Senden auf einem Channel vor dem dazugehörigen Receive auf dem Channel synchronisiert wird und dass das k-te Receive auf einem Channel mit Kapazität C vor der Beendigung des $k + C$-ten Send auf dem Channel synchronisiert wird [10]. In der Praxis verhält sich der Buffer eines Channels allerdings wie eine FIFO-Queue, das heißt, es wird bei einem Receive immer die ältest in dem Buffer vorhandene Nachricht ausgegeben. Im Folgenden wird daher immer von diesem Verhalten ausgegangen.

Sowohl bei gebufferten als auch bei ungebufferten Channels kann es zu Situationen kommen, in denen mehrere Routinen gleichzeitig auf dem selben Channel auf eine Nachricht warten. Wird nun auf diesem Channel gesendet wird eine der Routinen pseudo-zufällig ausgewählt, um die Nachricht zu empfangen.

Go macht es mit der Select-Operation möglich, auf die erste von mehreren erfolgreichen Channel-Operationen gleichzeitig zu warten. Ein Beispiel für solch ein Select befindet sich in Abb. 1. In diesem Beispiel besitzt die Select-Operation 3 Cases und einen Default-Case. Die

```
1  x := make(chan int)
2  y := make(chan int)
3  z := make(chan int)
4
5  select {
6    case <- x:
7      fmt.Println("x")
8    case a := <- y:
9      fmt.Println(a)
10   case  z <- 5:
11     fmt.Println(5)
12   default:
13     fmt.Println("default")
14 }
```

Abb. 1.: Beispielprogramm für Select

Cases bestehen aus verschiedenen Channel-Operationen (Receive auf Channel, Receive auf Channel mit direkter Variablendeklaration und Send auf Channel). Das Select-Statement probiert nun die Cases in einer zufälligen Reihenfolge aus. Findet es einen Case, in dem die Operation ausgeführt werden kann, wird die Operation, sowie der darunter stehende Block (Print-Statement) ausgeführt. Der Default-Case wird ausgeführt, wenn keiner der anderen Cases ausgeführt werden kann. Er ist allerdings nicht notwendig. Besitzt das Select-Statement keinen Default-Case, dann blockiert die Routine so lange, bis einer der Cases ausgeführt werden kann.

Channel können vorzeitig durch eine `close` Operation geschlossen werden. In diesem Fall kann auf diesem Channel nicht mehr gesendet werden. Versucht das Program auf einem geschlossenen Channel zu senden kommt es zu einem Laufzeitfehler und das Programm

wird abgebrochen. Versucht das Programm auf einem geschlossenen Channel zu lesen, wir ein Default-Wert zurückgegeben, ohne dass die Routine blockiert.

2.4. Betrachtete Probleme

Dieser Abschnitt betrachtet die Arten von Problemen, welche durch den Detektor erkannt werden sollen.

2.4.1. Mutex

Durch die Verwendung von Mutexen in nebenläufigen Programmen kann es zu sogenannten Deadlocks kommen. Blockieren sich dabei mehrere Routinen gegenseitig bezeichnen wir eine solche Situation als zyklisches Locking. Abb. 2 zeigt ein Beispiel, in welchem es zu zyklischem Locking kommen kann. Routine 0 und Routine 1 können dabei gleichzeitig

```
1  func main() {
2    var x sync.Mutex
3    var y sync.Mutex
4
5    go func() {
6      // Routine 1
7      x.Lock()
8      y.Lock()
9      y.Unlock()
10     x.Unlock()
11   }()
12
13   // Routine 0
14   y.Lock()
15   x.Lock()
16   x.Unlock()
17   y.Unlock()
18 }
```

Abb. 2.: Beispielprogramm zyklisches Locking

ausgeführt werden. Man betrachte den Fall, in dem Zeile 7 und 14 gleichzeitig ausgeführt werden, also Lock `y` von Routine 0 und Lock `x` von Routine 1 gehalten wird. In diesem Fall

kann in keiner der Routinen die nächste Zeile ausgeführt werden, da das jeweilige Locks, welches beansprucht werden soll bereits durch die andere Routine gehalten wird. Da sich diese Situation auch nicht von alleine auflösen kann, blockiert dass Programm, befindet sich also in einem zyklischen Deadlock.

Neben zyklischem Locking kann es auch durch doppeltes Locking von Mutexen zu Deadlocks kommen. Ein Beispiel dazu findet sich in Abb. 3. Der Mutex `x` soll hierbei mehrfach

```
1 func main() {
2   var x sync.Mutex
3
4   x.Lock()
5   x.Lock()
6 }
```

Abb. 3.: Beispielprogramm doppeltes Locking

geschlossen werden, ohne dass es zwischenzeitlich geöffnet wird. In diesem Fall blockiert die Routine endlos.

2.4.2. Channel

Es kommt auf Channels dann zu Problemen, wenn es Send oder Receive-Operationen gibt, welche keinen gültigen Kommunikationspartner besitzen. Der Detektor soll nun genau solche Programme erkennen, bei welchen es mögliche Ausführungspfade gibt, in denen solche Situationen auftreten können. Man betrachte das Programm in Abb. 4. Dieses besitzt zwei

```
1 func main() {
2   x := make(chan int, 0)
3
4   go func() { x <- 1}()   // 1
5   go func() { <- x }()    // 2
6   <- x                    // 3
7 }
```

Abb. 4.: Beispielprogramm mit hängendem Channel

Receive aber nur eine Send-Operation. Unabhängig von der Ausführung gibt es also eine

Receive-Operation, welche nie ausgeführt werden wird. Da ein Send oder Receive auf einem ungebufferten Channel, welche keinen Kommunikationspartner besitzt dazu führt, dass die entsprechende Routine ewig blockiert, führt dies somit zu einem Deadlock. Anders als in ungebufferten Kanälen müssen in gepufferten Kanälen Send und Receive einer Nachricht nicht gleichzeitig ablaufen. Hierbei kann es aber zu Situationen kommen, in dem eine Nachricht zwar erfolgreich gesendet, aber nie ausgelesen wird oder auf einem Channel öfter gelesen als gesendet wird. Es führt also entweder zu einer nie gelesenen Nachricht oder, wenn ein Receive ewig vor einem leeren Channel wartet ebenfalls zu einem blockierenden Deadlock. Da es sich in diesen Fällen nahezu immer um ungewünschtes verhalten handelt sucht der Detektor nach Situationen, in denen so etwas auftreten kann.

Ein weiterer möglicher kritischer Fehler bei Channels liegt in dem Senden auf einem geschlossenen Channel. Man betrachte dazu das Beispiel in Abb. 5. Das Programm versucht

```
1  func main() {
2    c := make(chan int)
3    go func() {
4      x <- 1
5    }()
6    go func() {
7      <- x
8    }()
9
10   close(x)
11 }
```

Abb. 5.: Beispielprogramm für Senden auf geschlossenem Channel

in Zeile 4 auf dem Channel `x` zu senden und in Zeile 7 zu empfangen. In Zeile 10 wird `x` geschlossen. Da alle drei Operationen nebenläufig Ablaufen ist es möglich, dass der Channel geschlossen wird, bevor das Senden abgeschlossen wird. Das Senden auf einem geschlossenen Channel führt dazu, dass das Programm abgebrochen wird. Das Empfangen auf einem geschlossen Channel ist hingegeben kein Problem. Hierbei wird lediglich ein Default-Wert ausgegeben, ohne das die Operation blockiert.

3. Detektor

Der Detektor läuft in zwei Schritten ab. Zuerst wird das Programm durchlaufen. Dabei wird ein Trace aufgezeichnet, welche die relevanten Informationen über den Programmablauf aufzeichnet. Dieser Trace wird im Anschluss analysiert, um Informationen über Probleme in dem Programm zu erhalten. Das Ganze wird gegebenenfalls mehrfach wiederholt, um eine breitere Abdeckung des Programmcodes zu erreichen.

3.1. Trace

Um ein Program analysieren zu können, wird der Ablauf eines Programmdurchlaufs (Trace) aufgezeichnet. Der grundlegende Aufbau des Trace basiert auf [6]. Er wird aber um Informationen über Locks erweitert. Der Trace wird für jede Routine separat aufgezeichnet. Außerdem wird, anders als in [6] ein globaler Program-Counter für alle Routinen und nicht ein separater Counter für jede Routine verwendet. Dies ermöglicht es, bessere Rückschlüsse über den genauen Ablauf des Programms zu erziehen. Die Syntax des Traces in EBNF gibt

sich folgendermaßen:

$$
\begin{array}{lcll}
T & = & '' \, [\, '', \, \{ \, U \, \}, \, '' \,] \, '' ; & \text{Trace} \\
U & = & '' \, [\, '', \, \{ \, t \, \}, \, '' \,] \, '' ; & \text{lokaler Trace} \\
t & = & signal(t,i) \mid wait(t,i) \mid pre(t,as) \mid post(t,i,x!) & \text{Event} \\
 & & \mid post(t,i,x?,t') \mid post(t,default) \mid close(t,x) & \\
 & & \mid lock(t,y,b,c) \mid unlock(t,y); & \\
a & = & x, \, ('' \, ! \, '' \mid '' \, ? \, ''); & \\
as & = & a \mid (\{ \, a \, \}, [\, '' \, default \, '' \,]); & \\
b & = & '' - '' \mid ''t'' \mid ''r'' \mid ''tr'' & \\
c & = & ''0'' \mid ''1'' &
\end{array}
$$

wobei i die Id einer Routine, t einen globalen Zeitstempel, x die Id eines Channels und y die Id eines Locks darstellt. Die Events haben dabei folgende Bedeutung:

- `signal(t, i)`: In der momentanen Routine wurde eine Fork-Operation ausgeführt, d.h. eine neue Routine mit Id i wurde erzeugt.

- `wait(t, i)`: Die momentane Routine mit Id i wurde soeben erzeugt. Dies ist in allen Routinen außer der Main-Routine das erste Event in ihrem lokalen Trace.

- `pre(t, as)`: Die Routine ist an einer Send- oder Receive-Operation eines Channels oder an einem Select-Statement angekommen, dieses wurde aber noch nicht ausgeführt. Das Argument as gibt dabei die Richtung und den Channel an. Ist $as = x!$, dann befindet sich der Trace vor einer Send-Operation, bei $as = x?$ vor einer Receive-Operation. Bei einem Select-Statement ist as eine Liste aller Channels für die es einen Case in dem Statement gibt. Besitzt das Statement einen Default-Case, wird dieser ebenfalls in diese List aufgenommen.

- `post(t, i, x!)`: Dieses Event wird in dem Trace gespeichert, nachdem eine Send-Operation auf x erfolgreich abgeschlossen wurde. i gibt dabei die Id der sendenden Routine an.

- `post(t, i, x?, t')`: Dieses Event wird in dem Trace gespeichert, nachdem eine Receive-Operation des Channels x erfolgreich abgeschlossen wurde. i gibt dabei die Id der sendenden Routine an. t' gibt den Zeitstempel an, welcher bei dem Pre-Event der sendenden Routine galt. Durch die Speicherung der Id und des Zeitstempels der sendenden Routine bei einer Receive-Operation lassen sich die Send- und Receive-Operationen eindeutig zueinander zuordnen.

- `post(t, default)`: Wird in einem Select-Statement der Default-Case ausgeführt, wird dies in dem Trace der entsprechenden Routine durch $post(t, default)$ gespeichert.

- `close(t, x)`: Mit diesem Eintrag wird das schließen eines Channels x in dem Trace aufgezeichnet.

- `lock(t, y, b, c)`: Der Beanspruchungsversuch eines Locks mit id y wurde gestartet. b gibt dabei die Art der Beanspruchung an. Bei $b = r$ war es eine R-Lock Operation, bei $b = t$ eine Try-Lock Operation und bei $b = tr$ ein Try-R-Lock Operation. Bei einer normalen Lock-Operation ist $b = -$. Bei einer Try-Lock Operation kann es passieren, dass die Operation beendet wird, ohne das das Lock gehalten wird. In diesem Fall wird c auf 0, und sonst auf 1 gesetzt. Das Trace-Element wird vor der abgeschlossen Beanspruchung in den Trace eingefügt um sicher zu stellen, dass ein zyklisches Locking auch dann erkannt wird, wenn es zu einem tatsächlichen Deadlock führt.

- `unlock(t, y)`: Das Lock mit id y wurde zum Zeitpunkt t wieder freigegeben.

Man betrachte als Beispiel das Programm in Abb. 6.
Dabei ergibt sich der folgende Trace:

$$[[signal(1,2),\ signal(2,3),\ signal(3,4),\ pre(4,a?,default),\ post(5,default)]$$
$$[wait(8,2),\ lock(9,1,-,1),\ pre(10,x!),\ post(16,2,x!),\ unlock(17,1)]$$
$$[wait(11,3),\ pre(12,y!),\ post(13,3,y!),\ pre(14,x?),\ post(15,2,x?,10)]$$
$$[wait(6,4),\ pre(7,y?),\ post(18,3,y?,12)]]$$

```
1  func main() {
2    x := make(chan int)
3    y := make(chan int)
4    a := make(chan int)
5
6    var v sync.RWMutex
7
8    // Routine 1
9
10   go func() {                    // Routine 2
11     v.Lock()
12     x <- 1
13     v.Unlock()
14   }()
15   go func() { y <- 1; <-x }()   // Routine 3
16   go func() { <-y }()           // Routine 4
17
18   select {
19   case <-a:
20   default:
21   }
22 }
```

Abb. 6.: Beispielprogramm für Tracer

Aus diesem Trace lässt sich der relevante Ablauf des Programms vollständig rekonstruieren.

3.2. Analyze

Nach dem das Programm durchlaufen und der Trace aufgezeichnet wurde, wird dieser anschließend analysiert. Dabei werden für die Erkennung von Problemen mit Mutexen und Channels zwei verschiedene Ansätze verwendet.

3.2.1. Mutexe

Man betrachte zuerst die Erkennung von zyklischem Locking. Da solche Situationen nur in ganz besonderen Situation auftreten (in Abb. 2 müssen Zeilen 7 und 14 genau gleichzeitig ausgeführt werden, ohne dass Zeile 8 oder 15 ausgeführt werden), muss ein Detektor, welcher vor solchen Situationen warnen soll, nicht nur tatsächliche Deadlocks, sondern vor allem

potenzielle, also nicht tatsächlich aufgetretene Deadlocks erkennen. Die Erkennung der potenziellen Deadlocks basiert hierbei auf iGoodLock [4] und UNDEAD [5]. Dabei wird für jede Routine ein Lock-Baum aufgebaut. Jeder in der Routine vorkommende (RW-)Mutex m_i wird durch einen Knoten k_i in dem Baum repräsentiert. In diesem Baum gibt es nun eine gerichtete Kante von k_1 nach k_2, wenn m_1 das von der Routine momentan gehaltene Lock ist, welches zuletzt von der Routine geschlossen worden ist, währen das Lock m_2 geschlossen wird [11]. Ist es nun möglich, Knoten aus verschiedenen Bäumen, welche den selben Mutex repräsentieren so zu verbinden, dass der entstehende Graph einen Zyklus enthält, bedeutet dies, dass in dem Programm zyklisches Locking möglich ist. Man betrachte dazu das Programm in Abb. 7. In Abb. 8 sind die entsprechenden Lock-Bäume, sowie der enthaltene Zyklus graphisch dargestellt. Es sei allerdings festzuhalten, dass besonders bei der Verwendung von RW-Locks, nicht jeder Zyklus auch direkt zu einem potenziellen Deadlock führen kann. Wie solche False-Positives verhinder werden können, wird in dem Abschnitt zur Implementierung des Detektors (Abs. 5.3) noch genauer betrachtet.

Doppeltes Locking kann, anders als das zyklische Locking nur erkannt werden, wenn es tatsächlich auftritt. Dazu wird überprüft, ob es in einem Trace für eine der Routinen ein Paar von Lock-Elementen des selben Mutex gibt, ohne dass es zwischen den beiden ein Unlock-Element des Mutex gibt. Hierbei muss außerdem gelten, dass es sich nicht bei beiden Lock-Operationen um RLocks handelt und das die spätere Lock-Operation keine TryLock-Operation ist.

3.2.2. Channel

Man möchte nun Situationen auf Channels erkennen, in welchen eine Routine versucht eine Nachricht zu versenden oder zu empfangen, es aber keinen möglichen Kommunikationspartner gibt. Für ungebufferte Channels, sowie in manchen Fällen bei gebufferten Channels kann dies zu einer blockade der entsprechenden Routine führen. Befindet sich die Situation in der Main-Routine, dann kann es dazu führen, dass das gesamte Programm nicht terminieren kann. In Fällen wie z.B. in Abb. 9, in welcher eine Nicht-Main-Routine blockiert (das Receive auf `x`

```
1  func cyclicLockingExample {
2    var v Mutex
3    var w Mutex
4    var x Mutex
5    var y Mutex
6    var z Mutex
7
8    go func ( ) { // R1
9      v . Lock ( )
10     w . Lock ( )
11     w . Unlock ( )
12     v . Unlock ( )
13     y . Lock ( )
14     z . Lock ( )
15     z . Unlock ( )
16     x . Lock ( )
17     x . Unlock ( )
18     y . Unlock ( )
19   }()
20
21   go func ( ) { // R2
22     w . Lock ( )
23     x . Lock ( )
24     x . Unlock ( )
25     w . Unlock ( )
26   }()
27
28   go func ( ) { // R3
29     x . Lock ( )
30     v . Lock ( )
31     v . Unlock ( )
32     x . Unlock ( )
33   }
```

Abb. 7.: Beispielprogramm zyklisches Locking

findet keinen Send-Partner) wird die entsprechende Routine bei der Terminierung der Main-Routine automatisch abgebrochen. Man bezeichne einen solchen Fall als hängende Routine. Mit gebufferten Channels kann es passieren, das eine Send-Operation zwar erfolgreich ausgeführt wird, die entsprechende Nachricht aber nie ausgelesen wird.

Um solche Situationen zu erkennen werden alle möglichen Paare von Kommunikationspartner betrachtet, und dabei überprüft, ob diese Zuordnung zu Send- oder Receive-Operationen führt, welche keine möglichen Partner besitzen. Dabei sei zu beachten, dass nur weil eine Send- und eine Receive-Operation auf dem selben Channel geschehen, nicht in jedem Fall

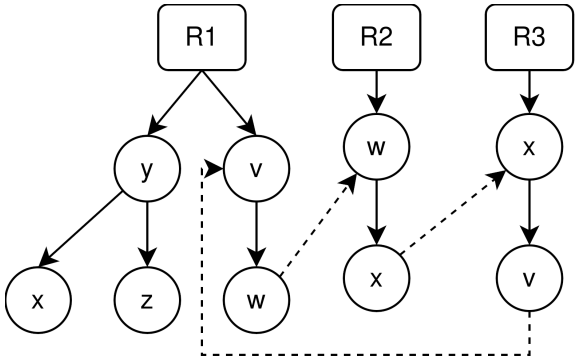

Abb. 8.: Graphische Darstellung des Lock-Graphen für das Beispielprogramm in Abb. 7. Durch die gestrichelten Pfeile wird der enthaltenen Zyklus angezeigt.

```
1 func main() {
2   x := make(chan int, 0)
3   go func() { <- x }()
4 }
```

Abb. 9.: Hängender Channel ohne Deadlock

eine Kommunikation zwischen diesen möglich ist. Man betrachte dazu das Beispiel in Abb. 10. Auf dem Channel `x` wird in 1 gesendet und kann in 2 und 3 empfangen werden.

```
1 func main() {
2   x := make(chan int)
3   y := make(chan int)
4   go func() { x <- 1; y <- 1 }   // 1
5   go func() { <- y; <- x }       // 2
6   <- x                           // 3
7 }
```

Abb. 10.: Beispielprogramm für unmögliche Synchronisation

Da es zwei Receive, aber nur eine Send-Operation gibt, kommt es zu einem hängenden Channel. Betrachtet man nur Channel `x` könnte man davon ausgehen, dass 1 nach 2 senden kann. Dies ist aber nicht möglich. Da der Channel `y` in 1 nach `x` sendet, in 2 allerdings von `x` empfangen muss, ist eine Synchronisierung auf `x` von 1 nach 2 nicht möglich. Die beiden Operationen bilden demnach keine möglichen Kommunikationspartner.

Um mögliche Kommunikationspartner zu erkennen, nicht mögliche Kommunikationspartner

wie in Abb. 10 aber auszuschließen, werden Vector-Clocks verwendet.
In einem ersten Durchlauf wird dabei der Trace mit Vector-Clock Informationen nach der Methode von Fidge [12] erweitert. Für jede Routine wird eine Vector-Clock gespeichert, welche für jede Routine einen Wert enthält. Zu Begin werden all diese Werte auf 0 gesetzt. Bei jedem Post-Event, sowohl für Send als auch Receive und für Signal und Wait Elemente wird der Wert der eigenen Routine in der lokalen Vector-Clock um eins erhöht. Bei einem Post-Receive und einem Wait Element wird die Vectorclock vc' betrachtet, welche in der sendenden Routine zum Zeitpunkt des Post-Send- bzw. Signal-Elements vorlag. Da ein Send- bzw. Signal-Event immer vor dem Receive- bzw. Wait-Element erzeugt wird, wurde die entsprechende Vectorclock in jedem Fall bereits bestimmt. Die Zuordnung der Trace-Elemente ist möglich, da der globale Counter bei einem Send an den Empfangenden Channel mitgesendet wird, und in dem entsprechenden Post-Receive-Element gespeichert wird. Bei einem Select-Statement wird nur derjenige Fall betrachtet, der auch tatsächlich ausgeführt wurde. Diese Send-Vector-Clock `vc'` wird nun mit der lokalen Vector-Clock `vc` der empfangenden Routine, bzw. der Wait-Routine `q` verrechnet und ersetzt diese. Dabei gilt

```
if vc[q] <= vc'[q] {
  vc[q] = 1 + vc'[q]
}
for i := 0; i < n; i++ {
  vc[i] = max(vc[i], vc'[i])
}
```

wobei n die Anzahl der Routinen ist.
Für alle andern Elemente, z.B. Pre usw., wird einfach die lokale Vector-Clock der Routine übernommen, ohne diese zu verändern. Da nun die Vector-Clocks zu jedem Zeitpunk bestimmt wurde, kann jedem Send- und Receive-Trace-Element eine Pre- und eine Post-Vector-Clock zugeordnet werden. Dabei handelt es sich um die Vector-Clocks, die bei Erzeugung des Pre- bzw. Post-Events in der Routine, in der die Operation ausgeführt wurde vorlag. Für hängende Operationen, bei denen kein Post-Element existiert, werden alle Werte

der Post-Vector-Clock auf max(Int32) gesetzt. Für Close gilt, dass die Pre-Vectorclock gleich der Post-Vectorclock ist. Andere Elemente werden in den VAT nicht aufgenommen, da sie für die anschließende Analyse nicht benötigt werden.

Basierend auf Vectorclocks gilt, dass x ist eine Ursache von y ist $(x < y)$ wenn gilt, dass

$$VC(x) < VC(y) \Leftrightarrow \forall z(VC(x)[z] \leq VC(y)[z]) \wedge \exists z'(VC(x)[z'] < VC(y)[z']), \qquad (1)$$

wobei $VC(x)[z]$ das z-te Element der Vectorclock von x bezeichnet. Man bezeichne x und y als unvergleichbar $(x \lesseqgtr y)$, wenn $x \not< y$ und $y \not< x$. Sind zwei Vector-Clocks unvergleichbar, sind sie unabhängig und die entsprechenden Operationen können somit gleichzeitig auftreten. Man erkennt also zwei Operationen, welche auf ungebufferten Channels eine mögliche Kommunikation durchführen können daran, dass sie auf dem selben Channel definiert sind, eine Send- und eine Receive-Operation definieren und dass ihre Pre- oder Post-Vector-Clocks unvergleichbar sind.
Um alternative Kommunikationspartner für Channel zu finden, werden also die Pre- und Post-Vector-Clocks der VAT-Elemente mit dem selben Channel verglichen um solche Paare zu finden.
Wenn sich eine Send- und Receive-Operation, welche bezüglich der Vector-Clocks zwar kommunizieren könnten so von Mutexen umschlossen werden, das die Operationen nicht gleichzeitig ausgeführt werden können, bilden sie keine gültige Kommunikation.

In gebufferten Channels ist nicht notwendig, dass Send und Receive gleichzeitig ausgeführt werden. Theoretisch ist es einem gebufferten Channel möglich bei einem Send mit jedem Receive auf dem selben Channel zu kommunizieren, welches nicht streng vor dem Send ausgeführt werden muss. In der Praxis geben sich allerdings Situationen in welchen eine Synchronisation zwischen Send- und Receive nicht möglich ist. Man betrachte dazu das Beispiel in Abb. 11. Auf den ersten Blick scheint es möglich, dass beide Send (1, 2) mit dem Receive kommunizieren könnten. Dies ist aber nicht der Fall. Es ist klar ersichtlich, dass 1 von 2 ausgeführt wird. Da der Buffer von gebufferten Channels in der Praxis als FIFO-Queues funktionieren, bedeutet dies, dass in 3 in jedem Fall die Nachricht aus 1

```
1  func main() {
2    x := make(chan int, 2)
3
4    x <- 1      // 1
5
6    go func() {
7      x <- 2    // 2
8    }()
9
10   <-x         // 3
11 }
```

Abb. 11.: Beispielprogramm für unmögliche Synchronisation auf gebuffertem Channel

empfangen wird. Allgemein kann man sagen, dass das n-te Send mit dem n-ten Receive kommuniziert. Das 1 vor 2 ausgeführt werden muss ist aus den Vectorclocks ablesbar. Um zu überprüfen, ob eine Kommunikation möglich ist, werden daher für jede Send s Operation die Anzahl der Send Operationen auf dem selben Channel bestimmt, welche streng vor s ausgeführt werden ($\#s_<$) sowie die Anzahl der Operationen, welche nebenläufig mit s ausgeführt werden ($\#s_{\not<}$). Das Selbe wird analog auch für jedes Receive r ($\#r_<, \#r_{\not<}$) bestimmt. Damit ein Send-Receive Paar $s - r$ auf einem gebufferten Channel nun basierend auf der FIFO-Queue möglich ist, also eine mögliche Kommunikation bilden muss gelten, dass

$$\#s_< \leq \#r_< + \#r_{\not<} \tag{2}$$

$$\#s_< + \#s_{\not<} \geq \#r_< \tag{3}$$

Die Motivation für diese Formeln wird klar, wenn man das Beispiel in Abb. 12 betrachtet. Man suche nun nach möglichen Kommunikationspartnern für das Send in 2. Da der Buffer eines Channels in Go praktisch wie eine FIFO-queue implementiert ist, ist es nicht möglich, dass 2 mit 5 kommuniziert, da dies das erste Receive ist, das Send in 1 aber vor dem in 2 ausgeführt wird. Dass diese Kommunikation nicht möglich ist, wird durch Formel 2 sicher gestellt. Auch die Kommunikation mit dem Receive in 8 ist nicht möglich. Nur die Send in 1 und 4 können vor dem Send in 2 ausgeführt werden. Das bedeutet, das das Send in 2 maximal das 3. Send in dem Programm darstellt. Dies bedeutet, dass für

```
1  func main() {
2    c := make(chan int, 5)
3
4    go func() {
5      c <- 1      // 1
6      c <- 1      // 2
7      c <- 1      // 3
8    }()
9
10   go func() {
11     c <- 1      // 4
12   }()
13
14   <- c          // 5
15   <- c          // 6
16   <- c          // 7
17   <- c          // 8
18 }
```

Abb. 12.: Beispielprogramm als Motivation für Formeln (2) und (3)

einen möglicher Kommunikationspartner maximal 2 andere Receives vor dem möglichen Kommunikationspartner ausgeführt werden dürfen. Da 8 aber in jedem Fall das 4. Receive darstellt, ist eine Kommunikation zwischen 2 und 8 nicht mögliche. Formel 3 sorgt dafür, dass solche Paare nicht fälschlicherweise als Kommunikationspartner gesehen werden.

Nachdem nun alle möglichen Kommunikationspartner ermittelt wurden, können die möglichen Ausführungspfade betrachtet werden, um zu überprüfen, ob diese zu Problemen führen. Der Detektor geht dazu alle vorhandenen Send-Operationen durch und ordnet Schrittweise jeder Operation eine Receive-Operation zu, so dass die Send- und Receive-Operation potenzielle Kommunikationspartner bilde und keine zwei Send-Operationen die selbe Receive-Operation besitzen. Es kann nun vorkommen, dass ein Teil der Send-Operationen bereits eine Receive-Operation zugeordnet bekommen haben, einer weiteren Send-Operation aber kein gültiges Receive zugeordnet werden kann. Dies bedeutet, dass wenn der Ablauf des Programm wie in den bereits zugeordneten Paaren geschieht, die Operation, welcher keine gültige Receive-Operation zugeordnet werden kann, keinen gültigen Kommunikationspartner besitzt. Dies bedeutet, dass es zu einem blockierten Programm, einer hängenden Routine, oder einer nicht ausgelesenen Nachricht in einem gebufferten Channel kommt. In diesem Fall

wird eine Warnung mit den entsprechenden Informationen ausgegeben. Das selbe wird auch umgekehrt durchgeführt. Es wird als versucht jeder Receive-Operation eine Send-Operation zuzuordnen. Auch hier führt eine Situation, in welcher keine gültige Zuordnung für eine der Operationen möglich ist zu einem Bug.
Nicht alle betrachteten Ordnungen bilden gültige Ordnungen. Da die Operationen innerhalb einer Routine sequenziell abgearbeitet werden ist eine Zuordnung nur dann gültig, wenn für jede Operation `p`, welche in der Ordnung betrachtet wird auch alle anderen Operationen in der selben Routine, welche vor `p` ausgeführt werden in der Zuordnung vorhanden sind.

Man betrachte als Beispiel das Programm in Abb. 13. Zuerst betrachte man den Fall, in

```
1  func main() {
2    x := make(chan int)
3
4    go func() {
5      x <- 1          \\ 1
6      <- x            \\ 2
7    }()
8
9    go func() {
10     x <- 1          \\ 3
11   }()
12
13   <-x               \\ 4
14 }
```

Abb. 13.: Beispielprogramm für die Betrachtung der Vector-Clocks

dem 1 mit 4 und dann 3 mit 2 synchronisiert. In diesem Fall erhält man folgenden Trace:

$$[[signal(1,2), signal(2,3), pre(3,1?), post(9,1,1?,6)]$$
$$[wait(5,2), pre(6,1!), post(7,1,1!), pre(8,1?), post(12,2,1?,10)]$$
$$[wait(4,3), pre(10,1!), post(11,2,1!)]]$$

Bei diesem Ablauf besitzen alle Operationen einen gültigen Kommunikationspartner. Es tritt also kein Bug auf. Aus diesem Trace lassen sich nun die Vector-Clocks für die einzelnen Operationen berechnen. Der Ablauf des Programs, welcher dem Trace entspricht, inklusive

der Vector-Clocks ist in Abb. 14 gegeben. Von diesen lassen sich nun die Pre- und Post-

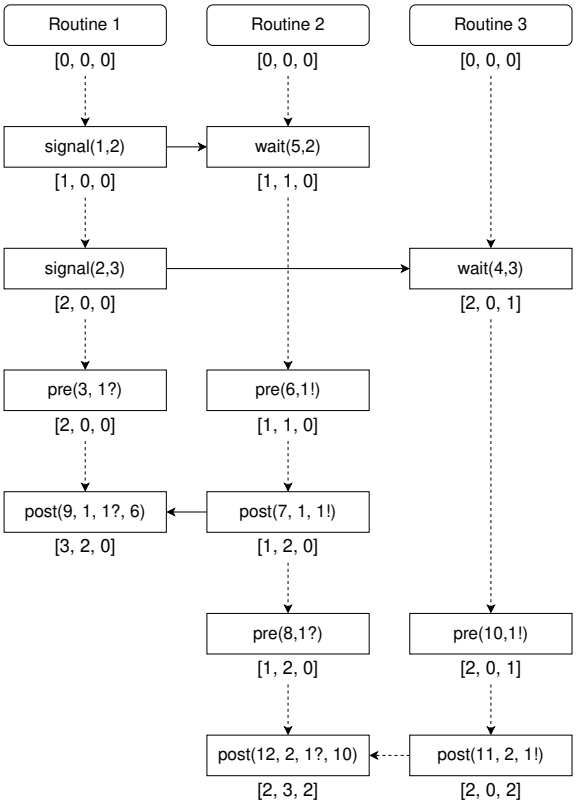

Abb. 14.: Ablaufdiagramm mit Vector-Clocks für das Beispiel in Abb. 13

Vector-Clocks direkt ablesen. Für den Vectorclock-Annotierte-Trace (VAT) werden diese in der Form ${}^{vc}a^{vc'}$ mit der Pre-Vector-Clock vc und der Post-Vector-Clock vc' und der Channel-Operation a angegeben. Der VAT gibt sich in diesem Fall also als

$$
\begin{aligned}
&[[^{[2,0,0]}x_4?^{[3,2,0]}] \\
&[^{[1,1,0]}x_1!^{[1,2,0]}, {}^{[1,2,0]}x_2?^{[2,3,2]}] \\
&[^{[2,0,1]}x_3!^{[2,0,2]}]]
\end{aligned}
$$

Basierend auf den Vectorclocks wird nun klar, dass 1 mit 4 und 3 sowohl mit 2 als auch 4 kommunizieren kann. Basierend darauf lassen sich nun für die von Send-Operationen

ausgehenden Ordnungen die Kommunikationen als

$$x_1! \rightarrow x_4?$$
$$x_3! \rightarrow x_2?$$

oder

$$x_3! \rightarrow x_4?$$
$$x_1! \rightarrow \lightning$$

und für die von Receive-Operationen ausgehenden Ordnungen als

$$x_4? \leftarrow x_1!$$
$$x_2? \leftarrow x_3!$$

oder

$$x_4? \leftarrow x_3!$$
$$x_2? \leftarrow \lightning$$

bestimmen. $\lightning$ bedeutet hierbei, das kein gültiger Kommunikationspartner möglich ist. Bei ungebufferten Channels führen solche Situationen zu einer blockierenden Routine oder gegebenenfalls sogar zur zu einer Blockades des gesamten Programms. Bei gebufferten Channels führen sie entweder zu einer Blockade oder zumindest zu einer nie gelesenen Nachricht. All diese Fälle werden als Bug betrachtet und werden somit von dem Detektor als potenzielle Bug angegeben.

Das betrachtete Beispiel zeigt gut den Vorteil der Verwendung von Vectorclocks. Bei dem Durchlaufen des Programms sind keine Bugs aufgetreten. Dennoch ist es mit den Vectorclocks möglich alternative Kommunikationsabläufe zu betrachten und potenzielle Bugs in diesen zu finden. Es zeigt außerdem die Notwendigkeit der Betrachtung von sowohl Pre- als auch

Post-Vectorclocks. Betrachtet man nur die Post-Vectorclocks, dann scheint es, as würde $x_1!$ streng vor $x_4?$ ausgeführt werden und somit keinen gültigen Kommunikationspartner für $x_1!$ bilden. Es ist aber leicht zu sehen, dass dies nicht der Fall ist. Durch die zusätzliche Betrachtung der Pre-Vectorclocks kann dies korrekt erkannt werden.

3.2.3. Close

Versuch ein Programm auf einem geschlossenen Channel zu senden kommt es zu einem Laufzeitfehler, welcher in einem Programmabbruch führt. Solch eine Situation lässt sich dadurch erkennen, dass eine Close- und eine Send-Operation auf dem selben Channel gleichzeitig ablaufen können, dass also die Pre- oder Post-Vectorclock der Send-Operation unvergleichbar mit der Vectorclock der Close-Operation ist. Situationen, bei denen die Vectorclocks der Send-Operation streng später als die der Close-Operation sind führen immer dazu, das es zu einem einem Send auf einem geschlossenen Channel kommt. In diesem Fall kommt es also immer zu einem Laufzeitfehler des Programms, durch welchen das Programm abgebrochen wird. Dies kann direkt, auch ohne die Betrachtung der Vectorclocks erkannt werden.

3.3. Select

In Go können Select-Statements dazu verwendet werden, abhängig davon, auf welchen Channels Nachrichten gesendet werden unterschiedliche Programmteile auszuführen. Dies erschwert die Analyse des Programs, da der tatsächliche Programmablauf dabei praktisch nicht-deterministisch werden kann [13]. Da der aufgezeichnete Trace immer nur einen Programmablauf betrachtet, führt dies zu Problemen, da nicht betrachtete Ausführungspfade zu Deadlocks oder anderen Problemen führen können. Eine Möglichkeit besteht darin, das Programm mehrfach auszuführen, und dabei immer unterschiedliche Select-Cases zu erzwingen. Ein solcher Ansatz wird unter anderem in GFuzz [7] verwendet. Dazu werden

die Select-Statements in dem Programmcode so verändert, dass aus den vorhandenen Cases eines gezielt bevorzugt werden kann. Der Ablauf eines Programms bezüglich seiner Select-Statements kann nun als Liste von Tupeln $[(s_0, c_0, e_0), \ldots, (s_n, c_n, e_n)]$ dargestellt werden. Dabei bezeichnet $s_i (0 \leq i \leq n)$ die ID einer Select-Operation, c_i die Anzahl der Cases in dieser Operation und e_n den Index des ausgeführten Case. Das Programm wird nun mehrfach durchlaufen, wobei die Ordnung zufällig verändert wird. Dazu wird nach jedem Durchlauf der Index e_i jedes Tupels auf einen zufälligen, aber gültigen Wert gesetzt. Da die Anzahl der möglichen Ausführungspfade durch die Select-Statements gegebenenfalls gegen unendlich gehen kann, ist es nicht möglich jede mögliche Kombination von Select-Cases auszuführen. Aus diesem Grund sammelt GFuzz während der Ausführung einer Ordnung Informationen über diese, um die Qualität einer Ordnung abzuschätzen. Aus diesen wird eine Wertung für die durchlaufenden Ordnung bestimmt, über welche die Anzahl der Mutationen bestimmt werden.

Die Betrachtung von verschiedenen Pfaden von Select wird auch für GoChan verwendet. Für jeden Durchlauf wird der Trace aufgezeichnet und dieser analysiert. Die Auswahl der Pfade wird dabei vereinfacht. Anders als in GFuzz basieren die betrachteten Ausführungsordnungen nicht auf schon versuchten Ordnungen, sondern es wird eine Menge von vollständig zufälligen Ordnungen betrachtet. Die Sammlung von Informationen über die einzelnen Abläufe bedeutet einen deutlichen Mehraufwand, sowohl bei der Implementierung, als auch bei der eigentlichen Bestimmung der für die Abschätzung benötigten Werte. Da der Nutzen dabei eher gering ist, wird darauf verzichtet.

4. Instrumentierung

Um die Analyse eines Programms durchzuführen, wird der Programmcode so verändert, dass die Aufzeichnung der Traces, sowie die Ausführung der Analysen automatisiert abläuft. Um einem Programmcode nicht von Hand verändern zu müssen wurde ein Instrumenter implementiert, welcher dies automatisch durchführt.

4.1. Trace

Wie bereits beschrieben, soll, um das Programm analysieren zu können, ein Trace aufgezeichnet werden.
Anders als in vielen anderen Programmen, welche den Trace von Go-Programmen analysieren, wie z.B. [14] oder [15] wird dabei der Tracer selbst implementiert und basiert nicht auf dem Go-Runtime-Tracer [16]. Dies ermöglicht es, den Tracer genau auf die benötigten Informationen zuzuschneiden und so einen geringeren negativen Einfluss auf die Laufzeit des Programms zu erreichen.

Um diesen Trace zu erzeugen, werden die Standardobjekte und Operationen auf Go durch Objekte und Operationen des Tracers ersetzt. Die Funktionsweisen dieser Ersetzungen sind im folgenden angegeben. Dabei werden nur solche Ersetzungen angegeben, welche direkt für die Erzeugung des Traces notwendig sind. Zusätzlich werden noch weitere Ersetzungen durchgeführt, wie z.B. die Ersetzung der Erzeugung von Mutexen und Channel von den

Standardvarianten zu den Varianten des Tracers. Dies werden in der Übersicht zur Vereinfachung nicht betrachte. Auch werden in der Übersicht nur die Elemente betrachtet, die für die Durchführung der Operation und dem Aufbau des Traces benötigt werden. Hilfselemente, wie z.B. Mutexe, welche verhindern, dass mehrere Routinen gleichzeitig auf die selbe Datenstruktur, z.B. die Liste der Listen, welche die Traces für die einzelnen Routinen speichern, zugreifen, werden nicht mit angegeben. Dabei sei c ein Zähler, nR ein Zähler für die Anzahl der Routinen, nM ein Zähler für die Anzahl der Mutexe und nC ein Zähler für die Anzahl der Channels. nM und nC werden bei der Erzeugung eines neuen Mutex bzw. eines neuen Channels atomarisch inkrementiert. Den erzeugten Elementen wird der neue Wert als id zugeordnet. All diese Zähler seien global und zu Beginn als 0 initialisiert. Außerdem bezeichnet mu einen Mutex, rmu einen RW-Mutex, ch einen Channel und B bzw. B_i mit $i \in \mathbb{N}$ den Körper einer Operation. Zusätzlich sei id die Id der Routine, in der eine Operation ausgeführt wird. $[signal(t, i)]^{id}$ bedeute, dass das entsprechende Element (hier als Beispiel $signal(t, i)$), in den Trace der Routine mit Id id eingeführt wird und $[+]^i$, dass in die Liste der Traces ein neuer, leerer Trace eingefügt wird, welcher für die Speicherung des Traces der Routine i verwendet wird. $\langle a|b\rangle$ bedeutet, dass ein Wert je nach Situation auf a oder b gesetzt wird. Welcher Wert dabei verwendet wird, ist aus der Beschreibung der Trace-Elemente in 3.1 erkennbar. e_1 bis e_n bezeichnet die Selektoren in einem Select statement. e_i^* bezeichnet dabei einen Identifier für einen Selektor, der sowohl die Id des beteiligten Channels beinhaltet, als auch die Information, ob es sich um ein Send oder Receive handelt. e_i^m bezeichnet die Nachricht, die in einem Case empfangen wurde.

go B	$\Rightarrow$	nr := atomicInc(nR); ts := atomicInc(c); [signal(ts, nr)]$^{\text{id}}$;
		[+]$^{\text{nr}}$; go { ts' := atomicInc(c); [wait(ts, nr)]$^{\text{nr}}$; B};
ch <- i	$\Rightarrow$	ts := atomicInc(c); [pre(ts, ch.id, true)]$^{\text{id}}$; ch <- {i, id, ts};
		ts' := atomicInc(c); [post(ts', ch.id, true, id)]$^{\text{id}}$
<- ch	$\Rightarrow$	ts := atomicInc(c); [pre(ts, ch.id, false)]$^{\text{id}}$;
		{i, id_send, ts_send} := <-c; ts' := atomicInc(c);
		[post(ts', ch.id, false, id_send, ts_send)]$^{\text{id}}$; return i;
close(ch)	$\Rightarrow$	ts := atomicInc(c); close(ch); [close(ts, ch.id)]$^{\text{id}}$
select($e_i \rightsquigarrow B_i$)	$\Rightarrow$	ts := atomicInc(c); [pre(ts, e_1^*, …, e_n^*, false)]$^{\text{id}}$;
		select($e_i \rightsquigarrow$ { ts' := atomicInc(c);
		[⟨post(ts, e_i.ch, false, e_i^m.id_send, e_i^m.ts_send) |
		post(ts, e_i.ch, true, id) ⟩]$^{\text{id}}$ B_i})
select($e_i \rightsquigarrow B_i$ | B_{def})	$\Rightarrow$	ts := atomicInc(c); [pre(ts, e_1^*, …, e_n^*, false)]$^{\text{id}}$;
		select($e_i \rightsquigarrow$ { ts' := atomicInc(c);
		[⟨post(ts, e_i.ch, false, e_i^m.id_send, e_i^m.ts_send) |
		post(ts, e_i.ch, true, id) ⟩]$^{\text{id}}$ B_i} |
		ts' := atomicInc(c); [default(ts)]$^{\text{id}}$; B_{def})
mu.(Try)Lock()	$\Rightarrow$	ts := atomicInc(c); [lock(ts, mu.id, ⟨-|t⟩, ⟨0|1⟩)]$^{\text{id}}$;
		mu.(Try)Lock();
mu.Unlock()	$\Rightarrow$	ts := atomicInc(c); mu.Unlock(); [unlock(ts, mu.id)]$^{\text{id}}$;
rmu.(Try)(R)Lock()	$\Rightarrow$	ts := atomicInc(c); rmu.(Try)(R)Lock();
		[lock(ts, rmu.id, ⟨-|t|r|tr⟩, ⟨0|1⟩)]$^{\text{id}}$;
rmu.RUnlock()	$\Rightarrow$	ts := atomicInc(c); rmu.RUnlock(); [unlock(ts, rmu.id)]$^{\text{id}}$;

Für Receive, Send und Close auf Channels, allen Operation auf Mutexen sowie der Erzeugung von Strukturen als Ersatz für die eigentlichen Mutexe und Channels sind Funktionen implementiert, welche die entsprechenden Operationen ersetzen und dabei sowohl die Aufzeichnung in dem Trace, als auch die eigentlichen Operationen durchführen. Hierbei sind besonders die Send- und Receive Operationen zu betrachten. Für den Trace muss dem Receive-Statement die Sender-Routine, sowie dessen momentane Zeitstempel bekannt sein.

Daher wird bei der eigentlichen Send-Operation nicht nur die eigentliche Information gesendet, sondern ein Objekt, welches die Information, sowie die Id der sendenden Routine und deren Zeitstempel beinhaltet.

4.2. Select

Neben den Ersetzungen der einzelnen Operationen, werden die Select-Statements zusätzlich verändert, um einen der Fälle bevorzugt auswählen zu können, wie in Kap. 3.3. beschrieben. Die Implementierung basiert dabei zum größten Teil auf [7]. Abb. 15 zeigt ein Beispiel für die Instrumentierung eines Select-Konstrukts. Die Select-Operation wird durch eine Switch-

```
1  select {
2    case <- Fire(time.Second):
3      Log("Timeout!")
4    case e := <- ch:
5      Log("Unexpected!")
6    case e := <- errCh:
7      Log("Error!")
8  }
```

```
1  switch FetchOrder(...) {
2    case 0:
3      select {
4        case <- Fire(time.Second):
5          Log("Timeout!")
6        case <- time.After(T):
7          .....
8      }
9    case 1:
10     select {
11       case e := <- ch:
12         Log("Unexpected")
13       case <- time.After(T):
14         .....
15     }
16   case 2:
17     select {
18       case e := <- errCh:
19         Log("Error")
20       case <- time.After(T):
21         .....
22     }
23   default:
24     .....
25 }
```

Abb. 15.: Beispiel für die Order-Enforcement-Instrumentierung eines Select-Statements vor (links) und nach der Implementierung (rechts). Ersetze ..... in dem Programm nach der Instrumentierung (rechts) durch das Programm vor der Instrumentierung (links). [7, gekürzt]

Operation auf der `Fetch-Order` ersetzt. Die Fetch-Order speichert für jede Select-Operation den Index des bevorzugten Case. Für eine gewisse, festgelegte Zeit T wird somit nur auf die in der `Fetch-Order` spezifizierten Operation gewartet. Wird diese in der vorgegebenen Zeit nicht ausgeführt, geht das Programm wieder in die Ausführung der ursprünglichen Select-Operation über um zu verhindern, dass es zu einer Blockierung kommt, welcher in dem originalen Code nicht vorgekommen wäre. Dasselbe gilt auch, wenn für die Select-Operation fälschlicherweise kein gültiges Case ausgewählt worden ist. Anders als in [7] beschrieben, wird für ein Select in dem selben Durchlauf immer der selbe Channel priorisiert, auch wenn die Operation mehrfach durchlaufen wird. Um für jede Ausführung einen eigenen Case zu priorisieren müsste, da die Ordnung bereits vor dem Durchlauf festgelegt werden soll, der Ablauf des Programms bereits bekannt sein oder eine unendliche Anzahl von ausgewählten Cases für jedes Select vorhanden sein. Dies ist allerdings nur möglich, wenn die Wahl der Cases der Select-Operationen keine Einfluss auf die Ausführung anderer Select-Cases hat. Dass dadurch nicht alle möglichen Abläufe durchlaufen werden können, muss dabei in Kauf genommen werden.

4.3. Automatisierter Instrumenter

Um den Trace zu erzeugen, müssen verschiedene Operationen durch Funktionen des Tracers ersetzt bzw. erweitert werden. Bei einem größeren Programmcode ist eine händische Instrumentierung nicht machbar. Da sich der Tracer auch negativ auf die Laufzeit des Programms auswirken kann, ist es in vielen Situationen nicht erwünscht, ihn in den eigentlichen Release-Code einzubauen, sondern eher in eine eigenständige Implementierung, welche nur für die Analyse verwendet wird. Um dies zu automatisieren wurde ein Programm implementiert, welches in der Lage ist, den Analyzer in normalen Go-Code einzufügen.

Der automatische Instrumenter besteht aus zwei Teilen. Zum einen werden alle in dem Programm vorhandenen ".go" Dateien instrumentiert. Zum anderen wird, basierend auf einem Template, eine neue Main-Datei erzeugt. Diese bestimmt die zu durchlaufenden

Ordnungen der Select-Operationen, führt diese nacheinander aus und konsolidiert die gefundenen Probleme.

4.3.1. Instrumentierung der Dateien

Bevor die einzelnen Dateien instrumentiert werden, wird der Ordner mit dem originalen Code durchlaufen, und seine Ordnerstruktur in den Output-Ordner kopiert. Anschließend werden alle Dateien durchlaufen. Handelt es sich nicht um ".go" Dateien werden sie einfach an die entsprechende Stelle im Output-Ordner kopiert. Handelt es sich um ".go" Dateien, dann wird der Programmcode so verändert, dass der erzeugte Code alle für den Analyzer notwendig Elemente enthält. Go besitzt in seiner Standart-Bibliothek ein Library zur Erzeugung und Bearbeitung des Abstract-Syntax-Trees einer Datei [17]. Zur Instrumentierung einer Datei wird dieser Baum mehrfach durchlaufen, dabei entsprechend angepasst und anschließend der veränderte AST als Code in der neuen Datei abgespeichert. Das mehrfache Durchlaufen ist notwendig, um Überschneidungen zwischen den einzelnen Teilen zu verhindern. Der Baum wird insgesamt drei mal durchlaufen.

1. Durchlauf: Im ersten Durchlauf werden der Import der GoChan-Bibliothek eingefügt und die Main-Funktion des Programms sowie Channels in Funktionsdeklarationen instrumentiert. Zuerst wird in jede Datei der Import der GoChan-Bibliothek eingeführt. Die Instrumentierung der Main-Funktion findet nur in der Main-Datei des Programms statt. Dabei wird der goChan-Analyzer initialisiert, und der Start der Analyse durch eine defer-Statement vorbereitet. In der Initialisierung des Analyzers werden die Datenstrukturen zur Speicherung des Traces erzeugt. Außerdem wird eine zusätzliche Routine mit einem Timer gestartet. Läuft dieser Timer ab (default: 20s), nimmt der Analyzer an, dass die Main-Routine blockiert, ohne dass es möglich ist, dass sich die Situation wieder auflöst und startet die Analyze, auch wenn das Programm noch nicht abgeschlossen ist. In diesem Fall wird das Programm automatisch abgebrochen. Für Programme, die von selbst eine längere Laufzeit haben muss die Dauer des Timers entsprechen angepasst werden. Anschließend

werden in allen Funktionen, welche Channels als Parameter oder Rückgabewerte haben, diese durch die entsprechenden Go-Chan-Objekte ersetzt. Außerdem wird die Main-Datei um ein globales Objekt zur Speicherung der Fetch-Order, also der bevorzugten Cases für die Select-Cases, erweitert. Dies wird nur getan, wenn das Programm mindestens eine Select besitzt.

2. Durchlauf Im zweiten Durchlauf werden die restlichen Channel Operationen instrumentiert. Hierbei werden alle Knoten des AST durchlaufen. Beinhaltet dieser eine Channel-Operation (z.B. Erzeugung eines Channels, Send, Receive usw.) wird sie durch die entsprechende Operation aus dem Go-Chan-Analyzer ersetzt. Dabei muss beachtet werden, dass solche Operationen in den verschiedensten Konstrukten (z.B. in defer, range usw.) enthalten sein können, welche jeweils einzeln betrachtet und ersetzt werden müssen. Der Channel selber wird dabei durch ein Objekt ersetzt, welches den eigentlichen Channel (allerdings auf Message-Objekt, welche sowohl die eigentliche Nachricht als auch Informationen über den Sender und dessen Zeitstempel besitzt), die Id des Channels, seine Kapazität sowie Informationen über die Position der Erzeugung des Channels im Code und ein Boolean zur Speicherung ob der Channel geschlossen wurde speichert. Die Definition dieses Struct wird, wie die Funktionen auf diesem Objekt durch GoChan definiert.

Auch die Erzeugung von neuen Go-Routinen wird hier instrumentiert. Ein Beispiel zur Instrumentierung der Erzeugung einer neuen Go-Routine findet sich in Abb. 16. `SpawnPre`

```
1 go func(i int) {
2     fmt.Println(i)
3 }(i)
```

```
1 func() {
2     GoChanRoutineIndex := goChan.SpawnPre()
3     go func(i int) {
4         goChan.SpawnPost(GoChanRoutineIndex)
5         {
6             fmt.Println(i)
7         }
8     }(i)
9 }()
```

Abb. 16.: Instrumentierung der Erzeugung einer neuen Go-Routine. Links: vor der Instrumentierung, rechts: nach der Instrumentierung.

erzeugt dabei das `signal` Element in dem Trace und gibt außerdem die Id der erzeugenden Routine zurück, welche in `SpawnPost` für die Erzeugung des `wait` Elements benötigt wird. Das umschließen des Ganzen mit einer weiteren Funktion erleichtert die Implementierung des Instrumenters, ist aber nicht unbedingt notwendig.

Zusätzlich werden hier auch die Select-Statements instrumentiert. Zum einen wird hier das Select-Statement wie in 3.3 beschrieben durch ein Switch-Statement mit mehreren Select-Statements ersetzt, um einen der Cases zu bevorzugen. Die Instrumentierung der Select-Statements für den Trace stellte dabei eine Komplikation da, da die Cases eines Select-Statements nur tatsächliche Channel-Operationen akzeptiert, diese also nicht direkt durch die Ersatzfunktionen des Analyzers ersetzt werden können. Die Lösung besteht darin, die Erzeugung der Events für den Tracer von dem eigentlichen Select-Case Statement zu trennen. Ein Beispiel dazu findet sich in Abb. 17. Zuerst wird die `goChan.PreSelect` Funktion aufgerufen, welche das Pre-Event in dem Trace erzeugt. Die Parameter dieser Funktion geben an, ob das Select-Statement einen Default-Case besitzt (in diesem Fall false), sowie für welche Channels es Cases gibt, und ob es sich bei diesen um Send- oder Receive-Statements handelt.
Anders als in dem Original-Code werden in dem instrumentierten Code auf Channels nicht nur die Nachrichten selbst, sondern auch Informationen über die sendende Routine verschickt. Da in einem Select-Case die Channel-Operationen nicht direkt ersetzt werden können, müssen nun für alle Cases, welche durch eine Send-Operation ausgelöst werden diese Nachrichten erzeugt werden. In dem Beispiel git es einen Case, auf den dies Zutrifft. Durch `sel_RAjWwhTH := goChan.BuildMessage(1)` wird diese erzeugt. `RAjWwhTH` ist dabei lediglich eine zufällige Zeichenkette, um unterschiedliche Nachrichten voneinander trennen zu können. Anschließend beginnt das Switch-Statement. `goChanFetchOrder` ist dabei die globale Map, die für jedes Select-Statement den für diesen Durchlauf bevorzugte Case angibt. `1` ist dabei die Id des Select. Auf den Cases werden nun normale Channel-Operationen ausgeführt. Da aber nicht nur die eigentlichen Nachrichten versendet werden, muss bei Receive-Statements mit anschließendem Assignment die eigentliche Nachricht aus dem Channel extrahiert werden. Außerdem wird bei einem Send-Statement die Nachricht durch

```
1 select {
2   case a := <-x:
3     fmt.Println(a)
4   case y <- 1:
5     fmt.Println(1)
6 }
```

```
1  goChan.PreSelect(false, x.GetIdPre(true),
2                   y.GetIdPre(false))
3  sel_RAjWwhTH := goChan.BuildMessage(1)
4  switch goChanFetchOrder[1] {
5  case 0:
6    select {
7    case sel_VlBzgbaiCM := <-x.GetChan():
8      x.Post(true, sel_VlBzgbaiCM)
9      a := sel_VlBzgbaiCM.GetInfo()
10     fmt.Println(a)
11   case <-time.After(time.Second):
12     ....
13   }
14 case 1:
15   select {
16   case y.GetChan() <- sel_RAjWwhTH:
17     y.Post(false, sel_RAjWwhTH)
18     fmt.Println(1)
19   case <-time.After(time.Second):
20     ....
21   }
22 default:
23   ....
24 }
25
26
27
28 Mit .... als:
29 select {
30   case sel_VlBzgbaiCM := <-x.GetChan():
31     x.Post(true, sel_VlBzgbaiCM)
32     a := sel_VlBzgbaiCM.GetInfo()
33     fmt.Println(a)
34   case y.GetChan() <- sel_RAjWwhTH:
35     y.Post(false, sel_RAjWwhTH)
36     fmt.Println(1)
37   }
```

Abb. 17.: Instrumentierung eines Select-Statements. Links: vor der Instrumentierung, rechts: nach der Instrumentierung.

die vorher Erzeugte ausgetauscht. Innerhalb des Cases wird die `goChan.Post` Funktion aufgerufen, um wenn der Case ausgewählt wurde das post-Event in dem Trace zu erzeugen. Neben der eigentlichen Instrumentierung der Select-Statements werden die Select- Cases mit ihren Ids außerdem aufgezeichnet. Beinhaltet das Programm mindestens ein Select-Statement, so wird, nachdem der Baum vollständig durchlaufen wurde die Main-Funktion so verändert, dass es Command-Line-Argumente annehmen kann. Dieses beinhaltet einen

String, in dem die `FetchOrder` für den Programmdurchlauf an des Programm übergeben werden kann.

3. Durchlauf Im dritten Durchlauf werden nun noch die Mutex-Operationen durch, von dem GoChan-Analyzer zur Verfügung gestellt Funktionen ersetzt, welche das eigentliche Locking als auch die Aufzeichnung des Traces übernehmen. Da es hierbei keine speziellen Strukturen wie select o.ä. gibt, ist eine einfache direkte Ersetzung der Operationen möglich. Die Objekte, welche die Mutexe dabei ersetzen, speichern dabei den eigentlichen Mutex, welcher das Locking tatsächlich ausführt, Informationen über die Position der Erzeugung des Mutex im Programmcode sowie die Id des Mutex. Für Mutex und RW-Mutex ist dabei jeweils ein eigenes Objekt definiert.

In Go müssen alle importierten Bibliotheken verwendet werden. Andernfalls kommt es zu einem Compiling-Error. Um zu verhindern, dass dies bei einer Datei, die weder Mutex noch Channel-Operationen besitzt geschieht, wird nach der abgeschlossenen Instrumentierung das "goimport" tool aufgerufen, welches nicht verwendete Imports automatisch entfernt.

4.3.2. Neue Main-Datei

Der in 4.3 erzeugt Code ist nun unter Angabe der bevorzugten Select-Cases prinzipiell lauffähig. Allerdings wird er nur einmal durchlaufen. Aufgrund der Select-Statements soll das Programm nun aber mehrfach, mit verschiedenen Select-Orders durchlaufen werden. Dazu wird eine neue Main-Datei erzeugt, mit welcher das eigentliche Programm gestartet wird. Diese Datei wird über ein Template erzeugt, mit welchem, basierend auf den Aufzeichnungen der instrumentierten Select-Cases die neue Datei erzeugt wird. Zuerst wird eine Map erzeugt, welche für jedes Select-Statement die Anzahl der Cases, also die Anzahl der Cases in dem Switch-Statements speichert. Anschließend werden die `FetchOrder` für die eigentlichen Durchläufe erzeugt. Dazu wird für jedes Select-Statement eine gültige Case-Id zufällig ausgewählt. Der so erzeugte Ablauf wird nun, wenn der selbe Ablauf nicht bereits erzeugt

worden ist, gespeichert. Dabei wird gezählt, wie oft ein Ablauf erzeugt worden ist, welcher bereits zuvor erzeugt worden ist. Erreicht dieser Wert einen vorgegebenen Maximalwert, dann nimmt das Programm an, dass genug verschiedene Abläufe erzeugt werden. Außerdem kann eine maximale Anzahl an Durchläufen festgelegt werden, um zu verhinder, dass die Dauer der Analyse zu lange wird, wenn sehr viele Select-Statements, bzw. sehr viele Select-Cases vorhanden sind. Anschließend wird das Instrumentierte Programm für jeden dieser Abläufe über einen `exec.Command` Befehl ausgeführt, wobei die Ordnung als Command-Line-Argument übergeben wird (also z.B. `exec.Command(./Examples/select/select "0,1")`, wobei hierbei für das Select mit Id 0 der Case 1 bevorzugt ausgewählt wird). Für das Programm wird nun der Trace aufgezeichnet und dieser anschließend analysiert. Die dabei für jeden Durchlauf gefundenen Probleme und Informationen werden gesammelt und am Ende gesammelt ausgegeben, wobei darauf geachtet wird, dass die selbe Information nicht mehrfach angegeben wird.

4.3.3. Laufzeit

Instrumenter Zuerst soll die Laufzeit des Instrumenters betrachtet werden. Es ist erwartbar, dass sich die Laufzeit linear in der Anzahl der Ersetzungen in dem AST, also der Anzahl der Mutex- und Channel-Operationen verhält. Dies bestätigt sich auch durch die Messung der Laufzeit des Programms (vgl. Abb. 18)

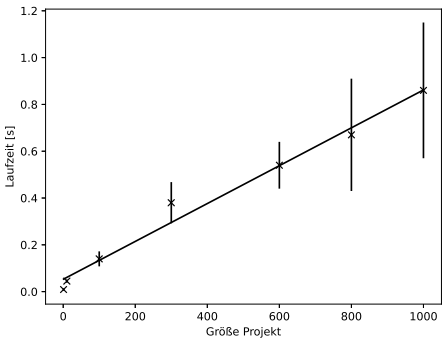

Abb. 18.: Laufzeit des Instrumenters

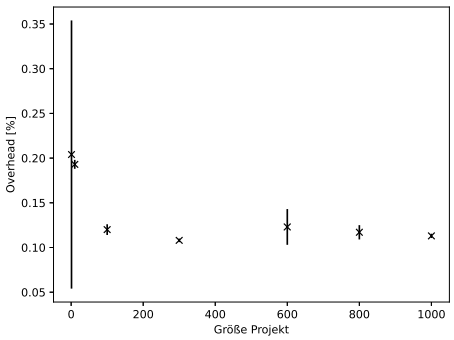

Abb. 19.: Prozentualer Overhead des Tracers ohne Analyse

Der abgebildete Graph zeigt die Laufzeit des Programms in s abhängig von der Größe des

Programms. Das Programm besteht dabei aus einem Testprogramm, welches alle möglichen Situationen mit Channels und Mutexen abbildet. Die Vergrößerung des Programmes wurde dadurch erreicht, dass die Datei mit dem Programmcode mehrfach in dem Projekt vorkam. Ein Projekt mit Größe n besteht vor der Instrumentierung also aus n Dateien, mit insgesamt $65n$ Zeilen von Code und $52n$ Ersetzungen in dem AST. Die tatsächliche Laufzeit des Instrumenters auf einen Programm hängt schlussendlich natürlich von der tatsächlichen Größe des Projekt und der Verteilung der Mutex- und Channel-Operationen in dem Code ab.

Zusätzlich wurde die Messung auch mit drei tatsächlichen Programmen durchgeführt. Die

Projekt	LOC	Nr. Dateien	Nr. Ersetzungen	Zeit [s]
ht-cat [18]	733	7	233	0.013 ± 0.006
go-dsp [19]	2229	18	600	0.029 ± 0.009
goker [8]	9783	103	4928	0.09 ± 0.03

Tab. 1.: Laufzeit des Instrumenters für ausgewählte Programme

dort gemessenen Werte befinden sich in Tabelle 1. Gerade in Abhängigkeit von der Anzahl der Ersetzungen, stimmen die hier gemessenen Werte mit denen in Abb. 18 gut überein, während es bei den anderen Parametern größere Abweichungen gibt. Dies bestätigt dass der dominante Faktor für die Laufzeit des Programms die Anzahl der Ersetzungen in dem AST ist, und die Laufzeit linear von dieser abhängt.

Tracer Folgend soll nun der Overhead des instrumentierten Codes im Vergleich zum originalen Code betrachtet werden. Hierbei wird nur die Laufzeit eines Durchlaufs des eigentlichen Programs (ohne Wiederholung aufgrund von Select), nicht aber der anschließenden Analyse betrachtet. Um den Overhead in Abhängigkeit von der Größe des Projektes messen zu können, wird das selbe Testprogramm betrachtet, welches bereits in der Messung für den Instrumenter verwendet wurde. Abb. 19 zeigt den gemessenen Overhead. Der durchschnittliche Overhead über alle gemessenen Werte liegt dabei bei 14 ± 2 %. Da der Overhead aber linear davon abhängt, wie groß der Anteil der Mutex- und Channel-Operationen im Verhältnis zu der Größe bzw. der Laufzeit des gesamten Programms ist, kann dieser Wert

abhängig von dem tatsächlichen Programm start schwanken. Dies wird unter anderem klar, wenn man den Overhead für ht-cat (9 ± 3 %) und go-dsp (60 ± 18 %) betrachtet, welche 51 ± 21 Prozentpunkte auseinander liegen.

5. Analyzer

Das folgende Kapitel beschreibt die Implementierung der Analyze des in Kap. 4.1 erstellten Trace.

5.1. Ablauf

Um das Programm zu Analysieren wird der instrumentierte Code ausgeführt. Besitzt er Select-Statements, wird er mehrfach durchgeführt, wobei jeweils eine andere, zufällige Ordnung für die Select-Cases betrachtet wird. Die betrachteten Ordnungen werden vor dem Durchlaufen der Programme erzeugt. Dabei kann vom Nutzer eine maximale Gesamtanzahl an Durchläufen festgelegt werden. Außerdem kann die Anzahl auch anhand der Dichte des Select-Cases beschränkt werden. Dazu wird eine neue, gültige Ordnung zufällig erzeugt und überprüft, ob diese Ordnung bereits zuvor erzeugt worden ist. Wenn sie neu ist, wird sie in die Liste der zu durchlaufenden Ordnungen eingefügt. Es wird dabei gezählt, wie oft eine Ordnung bereits zuvor erzeugt worden war. Auch für diesen Wert kann ein Maximalwert gesetzt werden.
Anschließend wird das Programm für jede der bestimmten Ordnungen analysiert. Da das Programm gegebenenfalls mehrfach analysiert wird ist es wahrscheinlich, dass die selben Probleme mehrfach erkannt werden. Es ist daher nicht sinnvoll die Informationen direkt nach jedem Durchlauf auszugeben. Daher werden die Fehler gesammelt, und dabei jeweils gespeichert, bei welchen Durchlaufen die Fehler aufgetreten sind. Nach Abschluss aller Durchläufe werden die gesammelten Fehler vollständig ausgegeben.

5.2. Aufzeichnung des Trace

Der Trace wird wie in Abschnitt. 4.1 beschriebenen erzeugt. Er wird dabei durch ein Slice von Slices gespeichert, wobei jeder Slice den Trace einer Routine speichert. Dazu wird für jedes Trace-Element ein Struct definiert, welches alle notwendigen Informationen speichert. Über ein Interface wird dafür gesorgt, dass diese Elemente alle in das Slice des Trace eingefügt werden können. Dabei wird über einen Mutex dafür gesorgt, dass nicht zwei Routinen gleichzeitig ein Element in den Trace einfügen können
Jeder Routine wird ein fortlaufender Wert eines atomaren Zählers als Id zugeordnet. Um die Id der Routine zu erhalten wird eine externe Bibliothek GoId [20] verwendet. Diese gibt eine interne Id für jede Routine aus, welche dann auf die Id der Routine gemappt wird. Auch die Ids für Channels und Mutexes werden durch laufende, atomare Zähler implementiert, welche in den Objekten für Channels und Mutexes gespeichert werden.

5.3. Mutex

Zuerst wird nach doppeltem Locking gesucht. Da ein doppeltes Locking immer zu einer Blockade führt, kann es nur auftreten, wenn das letzte Element in dem Trace einer Routine eine Lock- aber keine TryLock-Operation beschreibt. In diesem Fall wird der entsprechende Trace rückwerts durchlaufen, bis eine Operation auf dem selben Mutex gefunden wird. Handelt es sich dabei ebenfalls um eine Lock-Operation auf dem selben Mutex, bedeutet dies, dass ein Deadlock durch doppeltes Locking erkannt wurde. Dies ist allerdings nur der Fall, solange es sich nicht bei beiden Operationen um RLock-Operationen handelt. Wurde ein solcher Deadlock gefunden, wird eine eine entsprechende Nachricht zurückgegeben. Um potenzielle zyklische Deadlocks durch Mutexe erkennen zu können werden nun, wie in Kap. 2.4.1 beschrieben, Lock-Bäume verwendet. Diese werden basierend auf dem aufgezeichneten Trace aufgebaut. Dazu werden die Traces der einzelnen Routinen nacheinander durchlaufen. Für jede Routine wird eine Liste `currentLocks` aller Locks

erzeugt, die momentan von der Routine gehalten werden. Die einzelnen Elemente des Trace einer Routine werden nun durchlaufen. Handelt es sich dabei um ein Lock Event eines Locks `x`, wird eine s.g. `Dependendency` erzeugt und gespeichert. Diese beinhaltet das Lock `x` sowie eine Liste aller von der Routine momentan gehaltenen Locks, dem s.g. `holdingSet hs`. Dieses entspricht gerade `currentLocks`. Aus diesen `Dependencies` lassen sich also die Kanten in den Lockbäumen erschließen. Anschließend wird `x` in `currentLocks` eingefügt. Handelt es sich bei dem Element um ein Unlock-Event auf dem Lock `x`, dann wird das letzte Vorkommen von `x` auf `currentLocks` entfernt.

Nachdem der Trace einer Routine durchlaufen wurde, wird überprüft ob sich noch Elemente in `currentLocks` befinden. Ist dies der Fall, handelt es sich um Locks, welche zum Zeitpunkt der Terminierung des Programms noch nicht wieder freigegeben worden sind. Dies deutet darauf hin, dass die entsprechende Routine nicht beendet wurde, z.B. weil das Programm bzw. die Main-Routine beendet wurden. Dies kann einfach durch die entsprechende Logik des Programms zustande gekommen sein, es kann aber auch auf einen tatsächlich auftretenden Deadlock. In diesem Fall wird eine Warnung ausgegeben.

Ein potenzieller Deadlock gibt sich nun, wenn in diesem aus den Bäumen zusammengesetzten Graph ein Zyklus existiert. Nicht alle Zyklen bilden dabei gültige Zyklen. Zum Beispiel muss darauf geachtet werden, dass nicht alle Kanten durch die selbe Routine erzeugt wurden, und dass in zwei, in dem Kreis hintereinander folgende Kanten der gemeinsame Knoten nicht beides mal durch eine R-Lock Operation durch Kanten verbunden wurde. Gültige

Zyklen lassen sich durch dir folgenden Formeln charakterisieren.

$$\forall\ i, j \in \{1, ..., n\}\ \neg(hs_i \cap hs_j = \emptyset) \rightarrow (i = j) \tag{5.3.a}$$

$$\forall\ i \in \{1, ..., n-1\}\ mu_i \in hs_{i+1} \tag{5.3.b}$$

$$mu_n \in hs_1 \tag{5.3.c}$$

$$\forall i \in \{1, \ldots, n-1\}\ read(mu_i) \rightarrow (\forall mu \in hs_{i+1}(mu = mu_i) \rightarrow \neg read(mu)) \tag{5.3.d}$$

$$read(mu_n) \rightarrow (\forall\ mu \in hs_1 : (mu = mu_n) \rightarrow \neg read(mu)) \tag{5.3.e}$$

$$\begin{aligned}\forall\ i, j \in \{1, ..., n\}\ \neg(i = j) \rightarrow (\exists\ mu_1 \in hs_i\ \exists\ mu_2 \in hs_j((mu_1 = mu_2) \rightarrow \\ (read(mu_1) \wedge read(mu_2))))\end{aligned} \tag{5.3.f}$$

Dabei bezeichnet mu_i den Mutex und hs_i das `holdingSet` der i-ten Lock-Operation in dem Zyklus. (5.3.a) stellt sicher, dass das selbe Lock nicht in dem HoldingSet von zwei verschiedenen Routinen auftauchen kann. (5.3.b) und (5.3.c) sorgen dafür, dass es sich bei der Kette tatsächlich um einen Zyklus handelt, dass also das Lock einer Dependency immer in dem HoldingSet der nächsten Dependency enthalten ist und das Lock der letzten Routine wiederum im HoldingSet der ersten Routine liegt, um den Zyklus zu schließen. (5.3.d) bis (5.3.f) beschäftigen sich mit dem Einfluss von RW-Locks auf die Gültigkeit von Zyklen. Auch wenn (5.3.a) bis (5.3.c) erfüllt sind, ist dies dennoch keine gültige Kette, wenn sowohl der Mutex mu_i als auch der Mutex mu in hs_{i+1}, für das $mu = mu_i$ gilt, beides Reader-Locks sind, also Locks welche durch eine RLock-Operation erzeugt worden sind. Dass solche Pfade ausgeschlossen werden wird durch (5.3.d) und (5.3.e) sichergestellt. (5.3.f) beschäftigt sich mit Gate-Locks. Dabei handelt es sich um Mutexe, welche mehrere Teile des Programmcodes, welche bei gleichzeitiger Ausführung zu einem Deadlock führen könnten so umschließen, dass sie nicht gleichzeitig ausgeführt werden können und somit das Deadlock verhindert wird. Die Regel besagt nun, dass wenn es einen Mutex gibt, der in den HoldingSets zweier verschiedener Dependencies in dem Pfad vorkommt, so müssen beide diese Mutexe Reader-Locks sein. Sind sie es nicht, handelt es sich um Gate-Locks, und der entsprechende Pfad kann somit nicht zu einem Deadlock führen.

Für die Suche nach solchen Zyklen wird eine Depth-First-Search auf den gesammelten

Dependencies ausgeführt. Dazu wird zuerst eine Dependency auf einen Stack gelegt. Der Stack entspricht immer dem momentan betrachteten Pfad. Anschließend werden schrittweise weitere Dependencies auf den Stack gelegt, wobei darauf geachtet wird, dass aus jeder Routine immer nur maximal eine Dependency auf dem Stack liegt. Bevor eine Dependency zu dem Stack hinzugefügt wird, wird überprüft ob der durch den Stack betrachtete Pfad einen gültigen Pfad bilden würde, ob also (5.3.a), (5.3.b), (5.3.d) und (5.3.f) immer noch gelten würden. Ist dies nicht der Fall, so wird die Dependency nicht auf den Stack gelegt. Werden die Regeln hingegen erfüllt, dann wird überprüft, um der Stack nun einen gültigen Zyklus enthält, also auch (5.3.c) und (5.3.e) gültig sind. In diesem Fall wurde ein potenzielles Deadlock gefunden, und dies ausgegeben. Andernfalls werden weiter Dependency auf dem Stack hinzugefügt. Dies wird wiederholt, bis es keine Dependency mehr gibt, die auf den Stack gelegt werden könnte. In diesem Fall werden per Backtracking Dependencies von dem Stack entfernt, so dass andere Pfade ausprobiert werden können. Dies wird so lange durchgeführt bis alle gültigen Kombinationen durchprobiert worden sind.

5.4. Channels

Die Analyse des Programs zur Erkennung und Beschreibung von durch Channels ausgelösten Problemen läuft in mehreren Schritten ab. Zuerst werden die Vectorclock-Informationen der einzelnen Operationen bestimmt und mit diesen ein vectorclock-annotated Trace (VaT) erzeugt. Basierend auf diesen sucht der Analyzer nach potenziellen Situationen, welche zu blockenden Message-Bugs oder nicht gelesene Nachrichten auf gebufferten Channels führen können. Zum Schluss wird nach Situationen gesucht, bei denen es zu einem Send auf einen geschlossenen Channel kommen kann, da solche Situationen zu Laufzeitfehlern führen, welche den Abbruch eines Programms zur Folge haben. Receives auf geschlossenen Channels werden nicht betrachtet, da diese lediglich einen Null-Wert zurückgeben und nicht blocken, somit also nicht zu Laufzeitfehlern führen.

5.4.1. Bestimmung des vectorclock-annotierten Trace (VAT)

Basierend auf dem aufgezeichneten Trace soll nun ein vectorclock-annotierter Trace (VAT) erzeugt werden. Dieser besteht aus einer Reihe von `vcn`'s, welche jeweils eine Send-, Receive- oder Close-Operation repräsentieren. Andere Operation, wie z.B. signal-wait werden zwar bei der Berechnung der Vectorclocks beachtet, allerdings nicht in den VAT aufgenommen, da sie für die weitere Analyse nicht benötigt werden. Ein `vcn` beinhaltet dabei die Channel-Id, die Routine auf welcher die Operation ausgeführt wurde, ob es sich um Send- oder Receive handelt (bei Close beliebig gesetzt), die Position der Operation im Programmcode sowie die Pre- und Post-Vectorclocks der Operation. Eine Close Operation wird dabei dadurch erkannt, dass die Pre- und Post-Vectorclocks übereinstimmen.

Bevor der eigentliche VAT erzeugt wird werden erst die Vectorclocks zu allen Zeitpunkten bestimmt. Dazu wird für jede Routine eine Vectorclock initialisiert. Anschließend werden die Elemente in der Reihenfolge durchlaufen, in der sie in den Trace eingefügt wurden, also aufsteigend sortiert nach dem Timestamp der Trace-Elemente. Für jeden Zeitpunkt wird nun eine Vectorclock berechnet, wobei für jeden Timestamp immer diejenige Vectorclock gespeichert wird, die der Vectorclock entspricht, auf welcher die entsprechende Operation ausgeführt wurde. Für die Berechnung der Vectorclocks werden signal-wait Paare wie das Senden einer Nachricht von signal nach wait betrachtet.
Für send (post) und Signal bzw. Receive (post) und Wait werden nun die Vectorclocks aktualisiert. Für Send und Signal wird lediglich der eigene Timestamp in der eigenen Vectorclock um eins erhöht. Für die Aktualisierung bei einem Receive oder Wait wird die Vectorclock zur Zeit von Send oder Signal benötigt. Da diese in jedem Fall vor dem Receive oder Wait erzeugt worden sind, wurden sie bereits berechnet. Da für die Receive-Elemente in dem Trace die Zeitstempel der Send-Operationen gespeichert sind, ist eine eindeutige Zuordnung der Send- und Receive-Statements möglich. Für die Signal- und Wait-Elemente ist jeweils die Id der neu erzeugten Routine gespeichert. Es ist also auch hier eine eindeutige Zuordnung möglich. Die Vectorclock der Send- bzw. Signal-Operation kann also immer eindeutig bestimmt werden. Da bei Send-Receive immer der Timestamp des Pre-Elements

mitgesendet wird, erhält man mit diesem zuerst die Vector-Clock des Pre-Elements. Die Vectorclock des Post-Elements, welche für die Aktualisierung der Receive-Vectorclock benötigt wird lässt sich aus dieser durch erhöhen des Wertes an der Stelle der Id der Send-Routine um 1 bestimmen. Die Vectorclock dann wie in Abschnitt. 3.2.2 beschrieben aktualisiert werden.
Für alle anderen Element, also alle Pre-Elemente, Close-Operationen und Mutex-Operationen werden die Vectorclocks nicht verändert.

Nach der Berechnung der Vectorclocks kann nun der VAT bestimmt werden. Dazu werden die Traces für die einzelnen Routinen durchlaufen. Bei jedem Pre- und PreSelect-Element wird eine `vcn` erzeugt. Dazu wird der restliche Trace der selben Routine durchlaufen um das zugehörige Post-Element zu finden. Wird diese gefunden wird das `vcn` erzeugt, wobei die Pre- und Post-Vectorclock über den Zeitstempel der Pre- und Post-Elemente aus der Liste der Vectorclocks übernommen wird. Wird kein Post-Element gefunden, handelt es sich also um ein hängendes Event, wird die Pre-Vectorclock auf die Vectorclock des Pre-Events und alle Elemente der Post-Vectorclock auf `maxInt`, als den maximal möglichen Wert, gesetzt. Bei der Erzeugung der `vcn` wird außerdem gespeichert, welche Mutexe von der entsprechenden Routine momentan gehalten werden.
Für Close-Elemente gibt es nur ein Element in dem Trace. Aus diesem Grund besitzt das Element nur eine Vectorclock. Pre- und Post-Vectorclock werden dabei auf die gleiche Vectorclock gesetzt.

5.4.2. Erkennung potenzieller Communication-Bugs

Basierend auf dem VAT können nun tatsächlich aufgetretene oder potenzielle Communication-Bugs erkannt werden. Dabei wird nach Channel-Operationen gesucht, welche bei bestimmten Abläufen keine gültigen Kommunikationspartner besitzen. Dies führt zu einem blockierenden Bug durch das Warten auf Sends oder Receives oder das Senden von Nachrichten auf gebufferten Channels, ohne das die Nachricht jemals ausgelesen wird.

Zuerst werden basierend auf dem VAT alle potenziellen Kommunikationspartner für Send-Receive Paare bestimmt. Zur Suche nach alternativen Kommunikationspartner von ungebufferten Channels werden alle Kombinationen von zwei Elementen in dem VAT betrachtet. Dabei werden all diejenigen Elemente verglichen, bei welchen beide Operationen auf dem selben Channel ausgeführt werden und eines der Elemente ein Send- und das andere eine Receive-Operation ist. Zwei Operationen werden als potenzielle Kommunikationspartner angesehen, wenn die Pre- oder Vectorclocks der beiden Operationen unvergleichbar sind und die Listen der zur Zeit der Ausführung von der Routine gehaltenen Mutexe keine gemeinsamen Mutexe beinhaltet.
Für den gebufferten Channel müssen Send- und Receive nicht gleichzeitig ausgeführt werden. Aus diesem Grund müssen die Vectorclocks nicht unvergleichbar sein. Um mögliche Kommunikationspartner zu erkennen werden daher die in Abschnitt 3.2.2 beschriebenen Werten für jedes Element in dem VAT ermittelt. Dazu werden alle Send- mit allen Send- und alle Receive- mit allen Receive-Operationen verglichen und dabei überprüft, ob die Pre- und Post-Vectorclocks eine Happens-Before Relation bilden, bzw. ob sie unvergleichbar und damit nebenläufig sind. Für jede Operation wird dabei gezählt, mit wie vielen anderen Operationen dies der Fall ist. Für die Bestimmung der potenziellen Kommunikationspartner wird nun für jedes Paar von Send-Receive-Operationen auf dem selben Channel überprüft, ob die Formeln (2) und (3) erfüllt. In diesem Fall wird angenommen, dass die beiden Operationen miteinander kommunizieren können.
Basierend auf diesen Informationen wird für jede Send- und jede Receive-Operation eine Liste von potenziellen Kommunikationspartnern erstellt.
Basierend auf diesen möglichen Kommunikationspartnern werden nun alle möglichen Kommunikationsabläufe betrachtet. Für die Betrachtung aller Abläufe wird Backtracking verwendet. Zuerst wird schrittweise rekursiv für jede Send-Operation in dem Trace eine potenzielle Receive-Operation als Kommunikationspartner in einen partiellen Kommunikationsablauf eingefügt, sofern die Receive-Operation einen gültigen Kommunikationspartner bildet. Ein Receive ist ein potenzieller Kommunikationspartner, wenn eine Kommunikation basierend auf dem VAT möglich ist und die Receive-Operation noch von keiner anderen Send-Operation

als Partner verwendet wird. Besitzt ein solcher Pfad einen Kommunikationspartner für jedes Send-Statement in dem Trace, bedeutet dies, dass der entsprechende Ablauf nicht zu einem Bug führen kann. Ist es hingegen nicht möglich einem Send ein gültiges Receive zuzuordnen, dann führt der entsprechende Ablauf zu einem potenziellen Bug. Bevor dieser zurückgegeben werden kann muss erst noch überprüft werden, ob es sich um eine gültige Ausführungsordnung handelt, wie in Abschnitt 3.2.2 beschrieben. Ist diese der Fall, wird eine entsprechende Nachricht ausgegeben, welche die Operation, welche zu dem Bug führt, sowie den partiellen Ausführungspfad, bei welchem der Bug auftreten kann, enthält.
In beiden Fällen, in denen das Hinzufügen weiterer Send-Operationen nicht mehr möglich ist wird nun Backtracking verwendet, um die weiteren Ausführungspfade zu betrachten. Da die Reihenfolge, mit welcher die Send-Statements in den partiellen Ausführungspfad eingefügt werden werden, insbesondere welche Operation als erstes betrachtet wird, Auswirkung auf die Detektion von Problemen haben kann, wird das ganze für jede zyklische Permutation der Send-Operationen wiederholt.
Das Ganze wird anschließenden ein zweites Mal ausgeführt, wobei die Rollen von Send und Receive hierbei vertauscht sind. Es wird also für jedes Receive eine Send-Operation gesucht.

5.4.3. Erkennung von potenziellem Senden auf geschlossenem Channel

Für die Suche nach Situationen, die dazu führen können, dass auf einem geschlossenen Channel gesendet wird, werden die Vectorclock aller in dem Trace vorhandenen Close-Operationen mit den Vectorclocks aller Send-Operationen auf dem selben Channel verglichen. Ist mindestens eine der beiden Vectorclocks unvergleichbar, dann nimmt das Programm an, das eine Send-Operation auf einen geschlossenen Channel möglich ist, und gibt eine entsprechende Warnung zurück. Es kommt auch zu einem Send auf einem geschlossenen Channel, wenn die Vectorclock der Close-Operation streng vor den Vectorclocks der Send-Operation sind. In diesem Fall kommt es aber in jedem Fall zu einem Send auf einen geschlossenen Channel und damit zu einem Laufzeitfehler, welcher durch den Detektor

aufgefangen und erkannt wird.

5.5. Beispiel

Zur Veranschaulichung der Ausgabe befindet sich ein Beispielprogramm mit der dazugehörigen Ausgabe in Anhang B.

6. Auswertung

Im Folgenden soll betrachtet werden, wie gut der beschriebene Detektor in der Lage ist, Situationen wie in Abschnitt 2.4 beschrieben zu erkennen. Dabei werden sowohl künstlich konstruierte Situationen, als auch tatsächliche Programme betrachtet. Für die Betrachtung tatsächlicher Programme werden Programme aus Goker [8] verwendet. Dieses besitzt eine Sammlung von Programmteilen mit Concurrency-Bugs aus 9 großen open-source Anwendungen wie z.B. Kubernetes und Moby. Beschreibungen der Probleme, sowie die Ergebnisse des Detektors befinden in Anhang A.

6.1. Standardprobleme

Für die Analyse wurden insgesamt 44 Standardprobleme betrachtet. Dabei handelt es sich um konstruierte Programme, welche bestimmte Situationen beinhalten, die durch den Detektor erkannt werden sollen, bzw. Situationen, welche mit einer problematischen Situation verwechselt werden könnten, ohne dass ein Problem auftreten kann. Dabei wurde überprüft, ob der Detektor in der Lage ist, das in dem Programm erhaltene Problem richtig zu erkennen, bzw. zu erkennen wenn die vorliegende Situation nicht zu einem Problem führen kann. Eine tabellarische Beschreibung der betrachteten Situationen, sowie der Ergebnisse des Detektors ist in Tab. 2 in Anhang A aufgeführt.

Von den 44 Programmen konnten 41 korrekt kategorisiert werden. Dabei bestehen 18 Probleme aus Problemen mit Mutexen, 19 aus Problemen mit Channel und 7 mit einem Mix

aus Mutexen und Channel. Abbildungen 20 bis 23 geben an, welcher Anteil der Betrachteten Standardprobleme korrekt erkannt wurde.

Für Programme, bei denen der Fehler auf die Verwendung von Mutexen basiert, konnten 17 der 18 Probleme richtig kategorisiert werden. Dies entspricht ca. 94.4%. 1 Problem bildet hierbei ein False-Negative. Hierbei wird die Existenz eines potenziellen Deadlocks durch die Verschachtlung mehrere Routinen verschleiert.
Bei den Programmen, bei welchen es durch Channel zu Problemen kommen kann, lag der Detektor bei 18 der 19 Programme (94.7%) richtig. Bei dem Programm, welches nicht korrekt erkannt wurde handelt es sich um ein False-Positive.
Bei Programmen, welche sowohl Mutexe als auch Channels verwenden liegt die Erfolgsquote bei 6 aus 7 (85.7%).
Insgesamt hat der Detektor für die betrachteten Programme eine Trefferwahrscheinlichkeit von 93.2%.
Es sei noch dazu gesagt, dass die betrachteten Programme immer so implementiert worden sind, dass die entsprechenden Situationen auch in dem Durchlauf auftreten. Es ist allerdings auch möglich, dass Situationen bei den Durchläufen nicht durchlaufen werden, z.B. wenn sie sich ein einem konditionellen Block (If) befinden, bei welchem die Bedingung während keinem der Durchläufe wahr wird. Da die entsprechenden Operationen somit nicht aufgezeichnet werden können, ist es demnach logischerweise auch nicht möglich, dass der Detektor die entsprechenden Situationen erkennt.

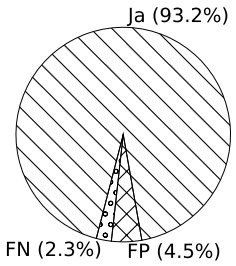

Abb. 20.: Verteilung der Ergebnisse für Standardprogramme

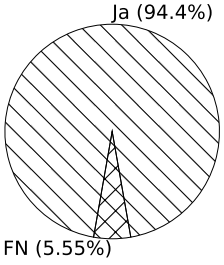

Abb. 21.: Verteilung der Ergebnisse für Standardprogramme mit Mutexen

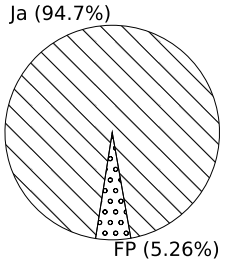

Abb. 22.: Verteilung der Ergebnisse für Standardprogramme mit Channel

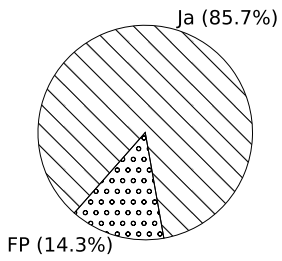

Abb. 23.: Verteilung der Ergebnisse für Standardprogramme mit Mutexen und Channel

6.2. GoKer

Für die Analyse wurden insgesamt 34 Programme betrachtet. Die betrachteten Programme und deren Ergebnisse befinden sich in Tab. 4 in Anhang A. Von den Programmen betrafen 15 die Verwendung von Resource wie Mutexen (14 korrekt erkannt), 11 die Verwendung von Kommunikationen (10 korrekt erkannt) und 8 eine Kombination aus Mutexen und Channel (7 korrekt erkannt). Da es sich hierbei nur um Situation handelt, welche tatsächlich Concurrency-Bugs enthalten ist eine Aufteilung nach FP und FN nicht möglich. Die Erfolgsraten des Detektors für die Programme aus GoKer (Abb. 24 bis 27) stimmen dabei in etwa mit denen der Standardprogramme überein. Für die Analyse wurden dabei nur solche Programme ausgewählt, welche basieren auf ihrer Beschreibung für den Detektor theoretisch erkennbare Situation enthielt. Situationen, welche sich auf andere Concurrency-Bugs, z.B. Race-Conditions bezogen, wurden nicht betrachtet. Die Betrachtung der Programme aus Goker hat einen Nachteil der hier verwendeten Methode, bzw. der Implementierung deutlich gemacht. Der Instrumenter ist nur in der Lage den vorliegenden Code zu instrumentieren. Es kann aber vorkommen, dass in einem Programm externe Funktionen verwendet werden, welche Mutexe oder Channel als Parameter oder Rückgabewerte besitzen. Da bei der Instrumentierung Mutexe und Channel durch eingens implementierte Objekte ersetzt werden, externe Funktionen aber nicht entsprechend Instrumentiert werden können kommt es hierbei zu Compiler-Fehlern. Die entsprechenden Programme sind daher nicht lauffähig

und können somit auch nicht analysiert werden. Programme aus GoKer, bei denen dies der Fall war wurden für die Analyse nicht betrachtet.

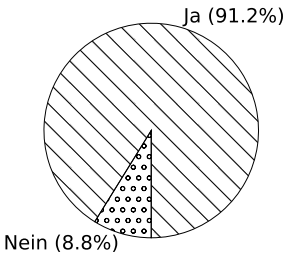

Abb. 24.: Verteilung der Ergebnisse für Programme aus GoKer

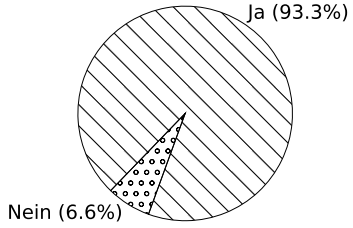

Abb. 25.: Verteilung der Ergebnisse für Programme aus GoKer mit Mutexen

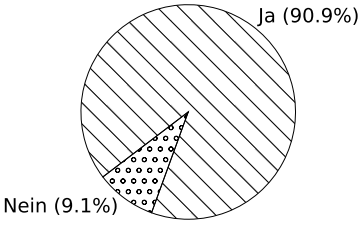

Abb. 26.: Verteilung der Ergebnisse für Programme aus GoKer mit Channel

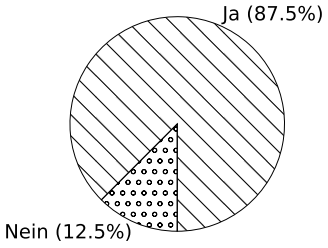

Abb. 27.: Verteilung der Ergebnisse für Programme aus GoKer mit Mutexen und Channel

6.3. Vergleich zu anderen Detektoren

Der Detektor vereinigt und erweitert Funktionsweisen aus verschiedenen Detektoren, und besitzt daher die Vorteile mehrerer Ansätze. Während sich einige Detektoren, wie z.B. UNDEAD [5] nur auf die Erkennung von Bugs, welche durch eine Art von Synchronisationsmechanismen ausgelöst werden spezialisiert, z.B. nur auf Mutexe, vereinigt der entwickelte Detektor Methoden zur Erkennung von Problemen durch sowohl Mutexe als auch Channels. Er ist außerdem, anders als z.B. GFuzz [7] nicht nur in der Lage Situationen zu erkennen, welche bei dem betrachteten Durchlauf tatsächlich zu einem blockenden Deadlock führen,

sondern ist auch in der Lage zu erkennen, wenn es in einem Programm durch eine andere Ausführungsordnung potenziell zu einem Bug kommen kann. Bei gebufferten Channels ist es hierbei auch in der Lage Kommunikationen zu erkennen, welche zwar nicht zu einem blockenden Bug führen, welche aber dennoch keine vollständig ausgeführte Kommunikation bilden.

6.4. Probleme und Verbesserungsmöglichkeiten

Der Detektor besitzt das selbe Problem wie alle dynamischen Ansätze. Er kann nur Probleme in den ausgeführten Programmteilen erkennen. Befinden sich tatsächlich oder potentielle Probleme teilweise oder vollständig in Bereichen, welche nicht ausgeführt wurden, z.B. auf Grund eines If-Statements, kann das entsprechende Problem nicht erkannt werden.

Das zweite größere Problem liegt in der Tatsache, dass die Kombination von Mutexen und Channels dazu führen kann, dass Bugs auftreten können bzw. dass Situationen, die bei der Betrachtung von nur Mutexen oder nur Channels zu Bugs führen könnten, welche durch die Kombination aber nicht auftreten können. Da Mutexe und Channel durch den Detektor getrennt voneinander betrachtet werden, können demnach False-Positives oder False-Negatives entstehen. Durch eine andere Vereinigung der Methoden, bzw. dem Betrachten der anderen Objekte in den Methoden wäre es prinzipiell möglich die Anzahl dieser Fehler zu reduzieren.

Als letztes geben sich außerdem Einschränkungen aus der Implementierung. Reine Go-Programme können einfach mit `go build` compiliert und anschließend das Executable direkt ausgeführt werden. Bei größeren Programmen ist dies häufig nicht direkt möglich. In diesen Fällen ist eine Anwendung des Detektors in seinem momentanen Stand nur sehr umständlich möglich. Ein anderes Problem besteht darin, dass importierte Libraries nicht instrumentiert werden. Dies bedeutet, dass Library-Funktionen, welche Channels oder Mutexe als Parameter oder Rückgabewerte besitzen mit der momentanen Instrumentierung zu Compiler-Fehlern führen. Um dies zu beheben wäre es möglich auch die entsprechenden

Bibliotheken zu instrumentieren und diese dann einzubinden. Dies ist momentan allerdings noch nicht (automatisiert) möglich.

7. Zusammenfassung

Ziel dieser Arbeit war es einen Detektor für von Mutexen und Channels erzeugte Concurrency-Bugs zu entwickeln und zu implementieren.
Der implementierte Detektor vereinigt und erweitert verschiedene Methoden zur Erkennung solcher Probleme.
Der Tracer und Analyzer wird dabei in den originalen Programmcode eingefügt. Dazu wurde ein Programm entwickelt, welches in der Lage ist diese Instrumentierung automatisch auszuführen.
Der Detektor erzeugt dynamisch einen Trace eines vorliegenden Programms, welcher im Anschluss analysiert werden kann.
Für Deadlocks durch Mutexe werden dabei Lock-Bäume zur Erkennung von zyklischem Locking verwendet, sowie nach doppeltem Locking gesucht.
Für Communication-Bugs, welche durch Channels erzeugt werden werden Pre- und Post-Vectorclocks bestimmt, welche eine Berechnung von potenziellen Kommunikationspartnern ermöglicht. Mit diesen lässt sich durch die Unordnung der Kommunikationspartner nach Programmabläufen suchen, bei welchen Kommunikationsoperationen kein gültigen Kommunikationspartner besitzen und daher den Code blockieren, oder bei gebufferten Channels Nachrichten versenden, welche nie gelesen werden.
Mit Hilfe der Vectorclocks lassen sich außerdem Situationen erkennen, in welchen ein Send auf einen geschlossenen Channel passieren könnte.
Zur Betrachtung von verschiedenen durch Selects verzweigte Ablaufpfade kann das Programm mehrfach durchlaufen werden, wobei jeweils ein anderer Ablauf bevorzugt werden

kann.

Bei der Anwendung des Detektors auf konstruierte und tatsächliche Programme ist er in der Lage gut 92% aller betrachteten Situationen richtig zu erkennen.

A. Beschreibung der betrachteten Programme

Id	Type	SubType	Ok?
1	Resource Deadlock	Zyklisches Locking	Ja
2	Resource Deadlock	Zyklisches Locking	Ja
3	Resource Deadlock	Zyklisches Locking	Ja
4	Resource Deadlock	Zyklisches Locking	Ja
5	Resource Deadlock	Zyklisches Locking	FN
7	Resource Deadlock	Zyklisches Locking	Ja
8	Resource Deadlock	Zyklisches Locking	Ja
11	Resource Deadlock	Zyklisches Locking	Ja
12	Resource Deadlock	Zyklisches Locking	Ja
13	Resource Deadlock	Zyklisches Locking	Ja
14	Resource Deadlock	Zyklisches Locking	Ja
15	Resource Deadlock	Zyklisches Locking	Ja
16	Resource Deadlock	Zyklisches Locking	Ja
6	Resource Deadlock	Doppeltes Locking	Ja
9	Resource Deadlock	Doppeltes Locking	Ja
10	Resource Deadlock	Doppeltes Locking	Ja
17	Resource Deadlock	Doppeltes Locking	Ja
18	Resource Deadlock	Doppeltes Locking	Ja

19	Communication Deadlock	Channel	Ja
20	Communication Deadlock	Channel	Ja
21	Communication Deadlock	Channel	Ja
22	Communication Deadlock	Channel	Ja
23	Communication Deadlock	Channel	Ja
24	Communication Deadlock	Channel	Ja
25	Communication Deadlock	Channel	Ja
26	Communication Deadlock	Channel	Ja
27	Communication Deadlock	Channel	Ja
28	Communication Deadlock	Channel	Ja
29	Communication Deadlock	Channel	Ja
30	Communication Deadlock	Channel	Ja
31	Communication Deadlock	Channel	Ja
32	Communication Deadlock	Channel	Ja
33	Communication Deadlock	Channel	Ja
34	Communication Deadlock	Channel	Ja
35	Communication Deadlock	Channel	Ja
36	Communication Deadlock	Channel	FP
37	Communication Deadlock	Channel	Ja
38	Mixed Deadlock	Channel&Mutex	Ja
39	Mixed Deadlock	Channel&Mutex	Ja
40	Mixed Deadlock	Channel&Mutex	FP
41	Mixed Deadlock	Channel&Mutex	Ja
42	Mixed Deadlock	Channel&Mutex	FP
43	Mixed Deadlock	Channel&Mutex	Ja
44	Mixed Deadlock	Channel&Mutex	Ja

Tab. 2.: Ergebnisse für die betrachteten Standartsituationen. Ja bedeutet, dass das Programm korrekt klassifiziert wurde. Bei FN handelte es sich um ein False-Negative und bei FP um ein False-Positive. Die Beschreibungen für die Situationen befinden sich in Tab. 3

Id	**Beschreibung**	**Ok?**
1	Potenzielles Deadlock durch zyklisches Locking von zwei Mutexen	Ja
2	Potenzielles Deadlock durch zyklisches Locking von drei Locks	Ja
3	Locking von zwei Locks, welche keinen Deadlock bilden, da Locking nicht zyklisch ist	Ja
4	Zyklisches Locking welches durch Gate-Locks nicht zu einem Deadlock führen kann	Ja
5	Potenzielles zyklisches Deadlock, welches durch Verschachtlung mehrerer Routinen (fork/join) verschleiert wird	FN
6	Tatsächliches Deadlock durch zyklisches Locking von Mutexen in zwei Routinen	Ja
7	Tatsächliches Deadlock durch zyklisches Locking von Mutexen in drei Routinen	Ja
8	Deadlock durch zyklisches Locking mit TryLock	Ja
9	Zyklisches Locking, welches durch TryLock nicht zu einem Deadlock führen kann	Ja
10	Potenzielles Deadlock mit RW-Mutexe in zwei Routinen	Ja
11	Kein potenzielles Deadlock mit RW-Mutexe in zwei Routinen	Ja
12	Kein potenzielles Deadlock, wegen Lock von RW-Locks als Gate-Locks	Ja
13	Potenzielles Deadlock, da R-Lock von Deadlock nicht als Gate-Lock funktioniert	Ja
14	Deadlock durch doppeltes Locken	Ja
15	Doppeltes Locking mit TryLock (TryLock → Lock)	Ja
16	Kein doppeltes Locking mit TryLock (Lock → TryLock)	Ja
17	Doppeltes Locking von RW-Locks, welches zu Deadlock führt (Lock→Lock, RLock→Lock, Lock→Rlock)	Ja

18	Doppeltes Locking von RW-Locks, welches nicht zu einem Deadlock führt(RLock→Rlock)	Ja
19	Deadlock oder hängende Routine durch Receive auf ungepuffertem Channel ohne Send	Ja
20	Deadlock oder hängende Routine durch Receive auf gepuffertem Channel ohne Send	Ja
21	Deadlock oder hängende Routiune durch 2-faches Receive mit 1-fachem Send [R_0: $\{\leftarrow x_1^0\}$, R_1: $\{x_1^0 \leftarrow 1\}$, R_2: $\{\leftarrow x_1^0\}$]	Ja
22	Deadlock oder hängende Routiune durch 2-faches Send mit 1-fachem Receive [R_0: $\{x_1^0 \leftarrow 1\}$, R_1: $\{x_1^0 \leftarrow 1\}$, R_2: $\{\leftarrow x_1^0\}$]	Ja
23	Deadlock oder hängende Routine durch Send auf ungepuffertem Channel ohne Receive	Ja
24	Kein Deadlock aber ungelesene Nachricht in Channel durch Send in Main-Routine auf gepuffertem Channel ohne Receive	Ja
25	Deadlock durch zweifaches Send auf gebufferten Channel in Kapazität 1 in Main-Routine ohne Receive	Ja
26	Ungelesene Nachricht bei [R_0: $\{c_1^1 \leftarrow 1; c_1^1 \leftarrow 1; c_1^1 \leftarrow 1\}$, R_1: $\{\leftarrow c_1^1\}$, R_2: $\{\leftarrow c_1^1\}$] sowie Erkennung der potenziellen Kommunikationspartner	Ja
27	Keine Probleme bei [R_0: $\{c_1^1 \leftarrow 1; c_1^1 \leftarrow 1\}$, R_1: $\{\leftarrow c_1^1\}$, R_2: $\{\leftarrow c_1^1\}$]	Ja
28	Kein Kommunikationspartner wenn Receive in Fork bei ungebuffertem Channel [R_0: $\{c_1^0 \leftarrow 1$; fork $R_1\}$, R_1: $\{\leftarrow c_1^0\}$]	Ja
29	Mögliche Kommunikationspartner wenn Receive in Fork bei gebuffertem Channel [R_0: $\{c_1^1 \leftarrow 1$; fork $R_1\}$, R_1: $\{\leftarrow c_1^1\}$]	Ja
30	Potenzielles Blocking bei anderer Communikationsordnung, ohne auftreten des Blocking bei Durchlauf [R_0: $\{\leftarrow c_1^0\}$, R_1: $\{c_1^0 \leftarrow 1; \leftarrow x_1^0\}$, R_2: $\{\leftarrow c_1^0\}$]	Ja

31	Deadlock bei Wahl eines bestimmten Select-Case [R_0: {$\leftarrow c_3^0$}, R_1: {$c_1^0 \leftarrow 1$}, R_2: {$c_2^0 \leftarrow 1$}, R_3: {select { case $\leftarrow c_1^0 \Rightarrow \{c_3^0 \leftarrow 1\}$, case $\leftarrow c_2^0 \Rightarrow \{\leftarrow c_3^0\}$}}]	Ja
32	Deadlock bei Wahl eines bestimmten Select-Case [R_0: {$\leftarrow c_3^0$}, R_1: {$c_1^0 \leftarrow 1$} R_2: {select { case $\leftarrow c_1^0 \Rightarrow \{c_3^0 \leftarrow 1\}$; default $\Rightarrow \{\leftarrow c_3^0\}$}}]	Ja
33	Tatsächliches Send auf geschlossenen Channel	Ja
34	Potenzielles aber nicht tatsächliches Send auf geschlossenen Channel Channel	Ja
35	Kein Problem, wenn Channel erst nach letztem Send geschlossen werden kann [R_0: {$c_1^0 \leftarrow 1$; close(c_1^1)}, R_1: {$\leftarrow c_1^1$}]	Ja
36	Kein Problem, wenn Channel erst nach letztem Send geschlossen werden kann [R_0: {$\leftarrow c_1^0$; close(c_1^0)}, R_1: {$c_1^1 \leftarrow 1$}]	FP
37	Korrekte Kommunikationspartner bei [R_0: {$c_1^1 \leftarrow 1$; $c_1^1 \leftarrow 1$; $c_1^1 \leftarrow 1$}, R_1: {$\leftarrow x$; $\leftarrow x$}] (letztes Send hat keinen Kommunikationspartner)	Ja
38	Deadlock, da gleichzeitiges Send und Receive durch Mutex Lock verhindert wird [R_0: {m_1.Lock; $\leftarrow c_1^0$; m_1.Unlock}, R_1: {m_1.Lock; $c_1^0 \leftarrow 1$; m_1.Unlock}]	Ja
39	Kein Problem, da gleichzeitiges Send und Receive durch RWMutex R-Lock nicht verhindert wird [R_0: {m_1^r.RLock; $\leftarrow c_1^0$; m_1^r.RUnlock}, R_1: {m_1^r.RLock; $c_1^0 \leftarrow 1$; m_1^r.RUnlock}]	Ja
40	Kein potenzielles zyklisches Locking da Operationen durch Channel-Operation getrennt sind [R_0: {$\leftarrow c_1^0$; m_1.Lock; m_2.Lock; m_2.Unock; m_1.Unlock;}, R_1: {m_2.Lock; m_1.Lock; m_1.Unlock; m_2.Unlock; $c_1^0 \leftarrow 1$}]	FP
41	Tatsächlicher Deadlock, da Send durch Lock nicht erreicht werden kann [R_0: {m_1.Lock; $c_1^0 \leftarrow 1$; m_1.Unlock}, R_1: {m_1.Lock; $\leftarrow c_1^0$; m_1.Unlock}]	Ja

42	Kein Blocking, da Send durch Lock nicht erreicht werden kann (R_2 vor R_1) [R_0: {m_1.Lock; m_2.Lock; m_2.Unlock; m_1.Unlock}, R_1: {select {case $\leftarrow c_1^0 \Rightarrow$ {m_2.Lock; m_1.Lock; m_1.Unlock; m_2.Unlock}, default $\Rightarrow$ {}}; R_2: {c_1^0 <- 1}}]	Ja
43	Potenzielles zyklisches Locking bei Wahl eines Select-Cases [R_0: {m_1.Lock; m_2.Lock; m_2.Unlock; m_1.Unlock}, R_1: {select {case $\leftarrow c_1^0 \Rightarrow$ {m_2.Lock; m_1.Lock; m_1.Unlock; m_2.Unlock}, default $\Rightarrow$ {}}; R_2: {c_1^0 <- 1}}]	Ja
44	Potenzielles zyklisches Locking bei Wahl eines DefaultSelect-Cases [R_0: {m_1.Lock; m_2.Lock; m_2.Unlock; m_1.Unlock}, R_1: {select {case $\leftarrow c_1^0 \Rightarrow$ {}, default $\Rightarrow$ {m_2.Lock; m_1.Lock; m_1.Unlock; m_2.Unlock}}; R_2: {c_1^0 <- 1}}]	Ja

Tab. 3.: Beschreibung der für die Auswertung betrachteten Standartsituationen. In den Fällen in denen Teile des Programmcodes angegeben sind sei R_0 die Main-Routine. c_i^j sei ein Channel mit Kapazität j, m_i ein Mutex und m_i^r ein RWMutex. Die Ergebnisse und befinden sich in Tab. 2

Id	Type	SubType	Ok?
Cockroach 584	Resource Deadlock	Doppeltes Locking	Ja
moby 17176	Resource Deadlock	Doppeltes Locking	Ja
moby 36114	Resource Deadlock	Doppeltes Locking	Ja
etcd 5509	Resource Deadlock	Doppeltes Locking	Ja
etcd 6708	Resource Deadlock	Doppeltes Locking	Ja
moby 7559	Resource Deadlock	Doppeltes Locking	Ja
serving 4829	Resource Deadlock	Doppeltes Locking	Ja
Cockroach 9935	Resource Deadlock	Doppeltes Locking	Ja
moby 4951	Resource Deadlock	Zyklisches Locking	Ja
Cockroach 7504	Resource Deadlock	Zyklisches Locking	Ja
Cockroach 10214	Resource Deadlock	Zyklisches Locking	Ja
hugo 3251	Resource Deadlock	Zyklisches Locking	Ja
kubernetes 13135	Resource Deadlock	Zyklisches Locking	Ja
Cockroach 6181	Resource Deadlock	RWR Deadlock	Ja
kubernetes 58107	Resource Deadlock	RWR Deadlock	FN
Cockroach 24808	Communication Deadlock	Channel	Ja
Cockroach 25456	Communication Deadlock	Channel	Ja
Cockroach 35073	Communication Deadlock	Channel	Ja
Cockroach 35931	Communication Deadlock	Channel	Ja
etcd 6857	Communication Deadlock	Channel	Ja
kubernetes 38669	Communication Deadlock	Channel	Ja
kubernetes 5316	Communication Deadlock	Channel	FN
kubernetes 70277	Communication Deadlock	Channel	Ja
moby 4395	Communication Deadlock	Channel	Ja
moby 29733	Communication Deadlock	Konditionelle Variable	Ja
moby 30408	Communication Deadlock	Konditionelle Variable	Ja
etcd 6873	Mixed Deadlock	Channel&Mutex	Ja

etcd 7902	Mixed Deadlock	Channel&Mutex	Ja
istio 16224	Mixed Deadlock	Channel&Mutex	Ja
kubernetes 10182	Mixed Deadlock	Channel&Mutex	Ja
kubernetes 1321	Mixed Deadlock	Channel&Mutex	FN
kubernetes 26980	Mixed Deadlock	Channel&Mutex	Ja
kubernetes 6632	Mixed Deadlock	Channel&Mutex	Ja
serving 2137	Mixed Deadlock	Channel&Mutex	Ja

Tab. 4.: Für Auswertung betrachteten Programme aus Gobench [8] sowie Information (Ok?) ob die Situation korrekt erkannt wurde. Ja bedeutet, dass die Situation korrekt erkannt wurde, FP bezeichnet ein False-Positive und FN ein False-Negative.

B. Beispielprogramm

Man betrachte das folgende Beispielprogramm.

```
1  func main() {
2    var m sync.Mutex
3    var n sync.Mutex
4
5    c := make(chan int)
6    d := make(chan int, 1)
7
8    go func() {
9      d <- 1
10     select {
11     case <-d:
12       close(c)
13     default:
14       <-c
15     }
16   }()
17
18   go func() {
19     m.Lock()
20     n.Lock()
21     n.Unlock()
22     m.Unlock()
23     <-c
24   }()
25
```

```
26      n.Lock()
27      m.Lock()
28      m.Unlock()
29      n.Unlock()
30      c <- 1
31  }
```

Dieses Programm besitzt den folgenden Output:

```
Determine switch execution order
Start Program Analysis
Analyse Program:   0%   1,1
Analyse Program:  50%   1,0
Analyse Program: 100%

Finish Analysis

Found Problems:

Found while examine the following orders:   1,1  1,0
Potential Cyclic Mutex Locking:
Lock: /home/.../output/show/main.go:109
    /home/.../output/show/main.go:108
Lock: /home/.../output/show/main.go:100
    /home/.../output/show/main.go:99


Found while examine the following orders:   1,1
No communication partner for receive at /home/.../output/show/main.go:103
  when running the following communication:
```

```
    /home/.../output/show/main.go:39 -> /home/.../output/show/main.go:41
    /home/.../output/show/main.go:112 -> /home/.../output/show/main.go:65

No communication partner for receive at /home/.../output/show/main.go:65
  when running the following communication:
    /home/.../output/show/main.go:112 -> /home/.../output/show/main.go:103
    /home/.../output/show/main.go:39 -> /home/.../output/show/main.go:41


Found while examine the following orders:   1,0
Possible Send to Closed Channel:
    Close: /home/.../output/show/main.go:48
    Send: /home/.../output/show/main.go:112
```

Die Pfade wurden dabei hier für eine bessere Lesbarkeit gekürzt.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] StackOverflow, "2020 developer survey," 2020. `https://insights.stackoverflow.com/survey/2020`.

[2] A. Gerrand, "Share memory by communicating," 2010. `https://go.dev/blog/codelab-share`.

[3] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," *Proc. FSE 2014*, pp. 155–165, 11 2014.

[4] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," *SIGPLAN Not.*, vol. 44, p. 110–120, jun 2009.

[5] J. Zhou, S. Silvestro, H. Liu, Y. Cai, and T. Liu, "Undead: Detecting and preventing deadlocks in production software," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (Los Alamitos, CA, USA), pp. 729–740, IEEE Computer Society, 11 2017.

[6] M. Sulzmann and K. Stadtmüller, "Two-phase dynamic analysis of message-passing go programs based on vector clocks," *CoRR*, vol. abs/1807.03585, 2018.

[7] Z. Liu, S. Xia, Y. Liang, L. Song, and H. Hu, "Who Goes First? Detecting Go Concurrency Bugs via Message Reordering," in *Proceedings of the 27th ACM International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '22, (New York, NY, USA), p. 888–902, Association for Computing Machinery, 2022.

[8] T. Yuan, G. Li, J. Lu, C. Liu, L. Li, and J. Xue, "Gobench: A benchmark suite of real-world go concurrency bugs," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 187–199, 2021. `https://github.com/timmyyuan/gobench`.

[9] The Go Team, "Effective go," 2022. `https://go.dev/doc/effective_go`.

[10] The Go Team, "The go memory model," 2022. `https://go.dev/ref/mem#chan`.

[11] R. Agarwal, L. Wang, and S. D. Stoller, "Detecting potential deadlocks with static analysis and run-time monitoring," in *Hardware and Software, Verification and Testing* (S. Ur, E. Bin, and Y. Wolfsthal, eds.), (Berlin, Heidelberg), pp. 191–207, Springer Berlin Heidelberg, 2006.

[12] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," 1988.

[13] The Go Team, "The go programming language specification - select," 2022. `https://go.dev/ref/spec#Select_statements`.

[14] S. Taheri and G. Gopalakrishnan, "Goat: Automated concurrency analysis and debugging tool for go," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 138–150, 2021.

[15] I. Danyliuk, "Visualizing concurrency in go," 2016. `https://divan.dev/posts/go_concurrency_visualize/`.

[16] The Go Team, "Go documentation: trace," 2022. `https://pkg.go.dev/cmd/trace`.

[17] The Go Team, "Go-documentation - ast." `https://pkg.go.dev/go/ast`.

[18] htcat, "htcat," 2015. https://github.com/htcat/htcat.

[19] mjibson, "go-dsp," 2018. https://github.com/mjibson/go-dsp.

[20] P. Mattis, "goid." https://github.com/petermattis/goid.